Big Pink #3
March 15, 1990

Big Pink #3 Table of Contents

1. Overview
    - 1.1. Pink Project Goals
    - 1.2. Pink System Architecture
    - 1.3. Pink Human Interface

2. System Objects
    - 2.1. System Foundation
        - 2.1.1. No Way (The Pink Run Time System)
        - 2.1.2. Utility Classes
        - 2.1.3. Cheetah (Persistent Objects)
        - 2.1.4. Credence (Concurrency Control & Recovery)
        - 2.1.5. Rainbow Warrior (Environment Variables)
        - 2.1.6. Babel (International Utilities)
        - 2.1.7. Tokens
    - 2.2. Application Framework
        - 2.2.1. Making Whoopee (The Application Framework)
        - 2.2.2. CHER (Document Architecture)
        - 2.2.3. ESP (Event Server)
        - 2.2.4. Scripting
        - 2.2.5. Donner Party (Collaboration Toolkit)
    - 2.3. Graphics
        - 2.3.1. Laser (Layer Server)
        - 2.3.2. Graphics Introduction
        - 2.3.3. Graphic Objects
    - 2.4. Printing
        - 2.4.1. The Outer Limits (Printing and Image Management)
        - 2.4.2. The Outer Limits (The Print Server)
        - 2.4.3. The Outer Limits (Printer Teams)
        - 2.4.4. InnerSpace (Scanning)
    - 2.5. Time
        - 2.5.1. Timing Services
        - 2.5.2. Time Ports and Sequences
    - 2.6. Sound
        - 2.6.1. Audio Objects
        - 2.6.2. Sound, Speech, and Telephony
        - 2.6.3. Editors
        - 2.6.4. Sound Effects
    - 2.7. Text
        - 2.7.1. Base Text Classes (Text Storage & Style Management)
        - 2.7.2. ZZText (Text Formating & Editing Classes)
        - 2.7.3. Text (Line Layout)

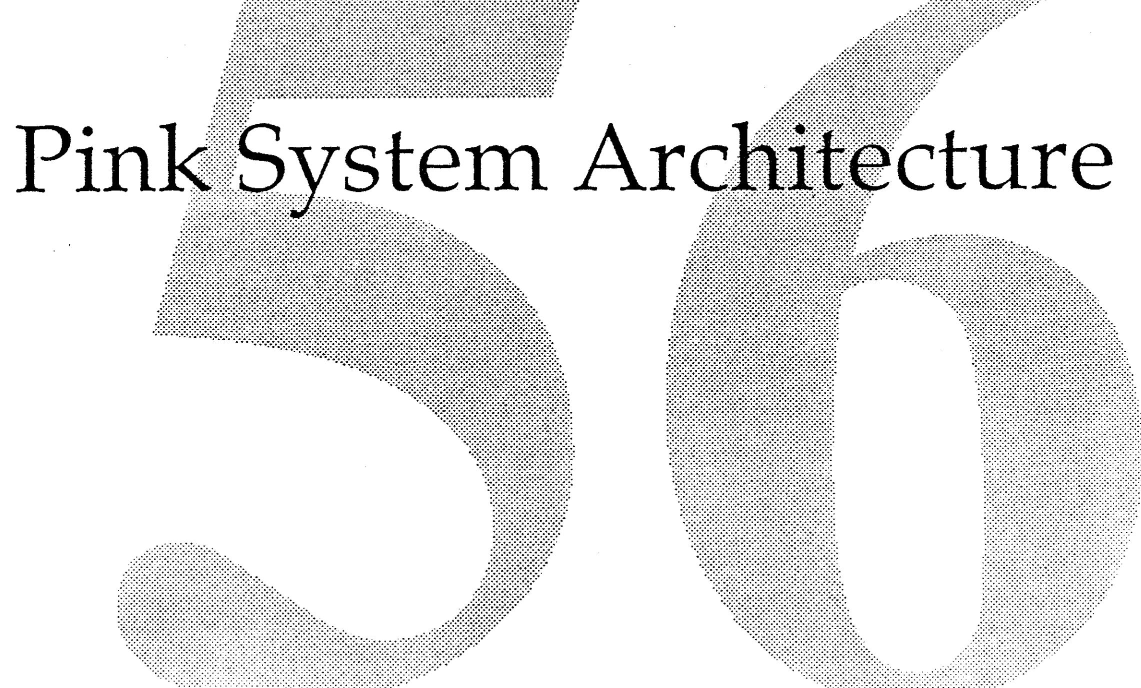
Big Pink #3
March 15, 1990

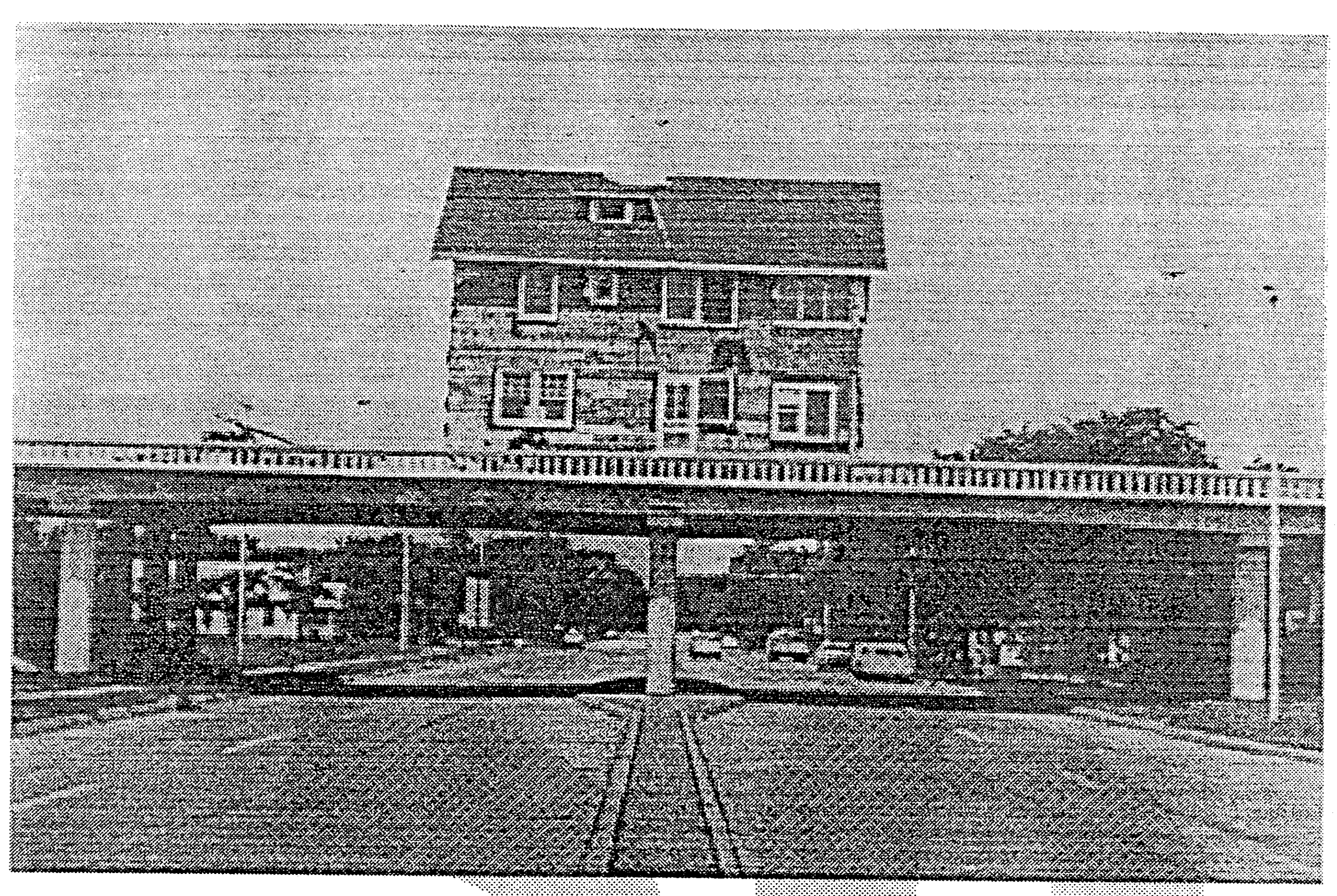
2.8. Files/Storage
  2.8.1. Psychokiller (Data Management)
  2.8.2. Pluto (File System)
  2.8.3. Bluto (Pink Personal AppleShare)
2.9. OS Services
  2.9.1. Opus/2 (Memory, Tasks, & IPC)
  2.9.2. Elixir (The Pink I/O System)
    2.9.2.1. Pink Booting Overview
    2.9.2.2. KT-22(Mass Storage I/O)
    2.9.2.3. Funnel of Love (NuBus I/O Framework)
    2.9.2.4. Rob Lowe(Video Framework)
2.10. Communication Services
  2.10.1. (Reserved)
  2.10.2. The Plumbing (Messages and Pipes)
  2.10.3. Scream (Server/Client Services)
  2.10.4. RedEye (Pink AppleMail™)
2.11. Network Services
  2.11.1. BabelFish (The Network)
  2.11.2. Coral Dawn (A/Rose)

3. Applications
  3.1. Valhalla (The Pink Finder)
    3.1.1. Thor (User Interface Specification)
    3.1.2. Odin (Programming Interface Specification)
  3.2. Blue Adapter
    3.2.1. Mood Indigo (The Blue Adapter)
    3.2.2. Scorpion (N&C Blue Adapter)
  3.3. Jane (An Advanced Word Processor)
  3.4. Tuffy (Pink Graphing Application)
  3.5. Hoops (A Human-Oriented Object Programming System)
  3.6. Online Documentation
  3.7. Don Quixote (UNIX adapter)

4. Project Issues
  4.1. Technical Documentation

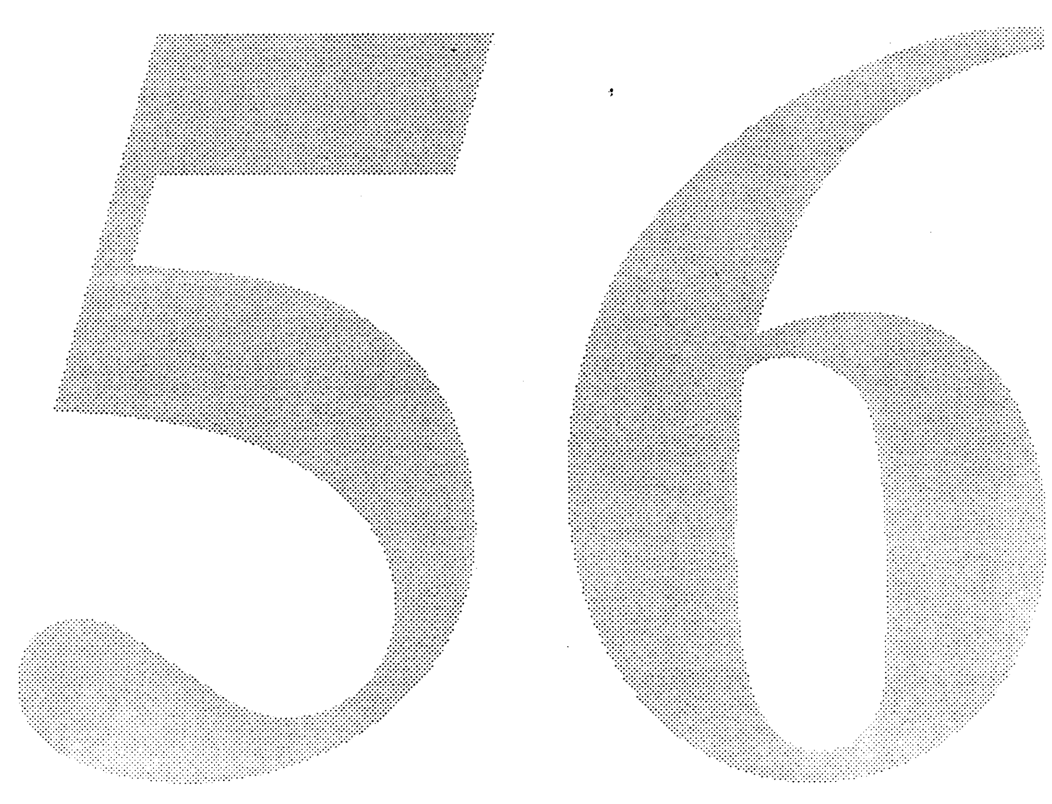
Pink System Architecture





Pink System Architecture

David Goldsmith



1. Introduction 1

2. Architectural Goals 1

2.1. Flexibility and Expandability 1

2.2. Portability 1

2.3. Performance 2

2.4. Robustness 2

2.5. Empower Developers 2

2.6. Support the Interface 3

2.7. A Global System 3

3. Architectural Principles 3

3.1. All Interfaces through Objects 3

3.2. Manage Commonality through Inheritance 4

3.3. Leverage Where Possible 4

3.4. Frameworks Protect Subsystems from Each Other 5

3.5. Let Resources Find You 5

4. Architectural Overview and Issues 6

4.1. The Opus/2 Kernel 6

4.2. The Run Time Environment 8

4.3. The Foundation Classes 9

4.4. "Operating System" Services 10

4.5. The Graphics System 11

4.6. Application Support 12

4.7. Text and International 14

4.8. Printing 15

4.9. Time, Sound, and Animation 16

4.10. Networking and Communications 16

4.11. The Desktop and Finder.. 17

4.12. Adapters.. 18

Introduction

Pink is Apple's new system software architecture. It draws on the strengths of Macintosh, but also introduces a new programming model and architecture designed to open up new opportunities for Apple and third party developers. This document gives an overview of the architecture of the Pink system.

Architectural Goals

Pink has several important architectural goals which are crucial to meeting the project goals of opening new opportunities for Apple and developers.

Flexibility and Expandability

Apple sometimes has difficulty meeting market opportunities because of the current Macintosh architecture. This is partly due to the many assumptions which were made by the original Macintosh team, which are deeply embedded in the architecture of the system. These assumptions make it difficult to add some new system features, since the features violate the assumptions on which the system was built. Worse, these assumptions are spread throughout the system, and are not easily changed.

Pink aims to be more flexible and expandable than the current Blue architecture, to enable Apple to more easily change and expand the system. Pink will need this flexibility to support platforms like Jaguar, and to grow to deal with systems beyond that.

No system is infinitely expandable: it is always aimed at a target. That target leads to design constraints, and sooner or later those constraints lead to obsolescence. Pink's goal is to extend the useful lifetime of the system and to ease the burden of modifying it.

The key to such flexibility is careful management of assumptions. Assumptions are a necessary evil: some assumptions must underlie every engineering project. The objective is to make no more assumptions than you must, and to isolate such assumptions wherever possible (the "mushroom" theory of programming). Pink achieves this objective through the use of objects.

There are two aspects of objects which help manage assumptions. The first is data abstraction: clients deal with an object through an abstract interface which deals only with the information that the client *must* know. Implementation details are hidden. The second aspect is type inheritance: clients deal only with the *level* of information they must know. Base classes provide the minimum protocol for dealing with an object; derived classes provide extra information for those who *need* to know.

Objects, of course, are not enough. Pink cannot meet its goal of flexibility without care and diligence on the part of all designers.

Portability

Apple does not currently have the option of moving its system software to another hardware platform. This is because it is written almost entirely in 68000 assembly language, driven by the need to fit in small amounts of memory (ROM and RAM). Although memory will never be free, our hardware has grown to the point where Apple can afford to trade off some memory consumption for the ability to run on platforms other than the 68000. Having this ability lets Apple take maximum advantage of competition among microprocessor vendors. This enables us to introduce important new products like Jaguar. Considering the agony of developing a complete suite of system software, the company's best interests demand we not tie Pink to a specific processor architecture.

"The only reasonable numbers are one and infinity." Thus, portability means more than portability to Jaguar. There will be other machines beyond Jaguar. Some day, the company might even want to run Pink on something really obscure, like an Intel processor ("bite your tongue!").

To meet this goal in Pink, all but the most performance critical code is being written in a processor- and system-independent fashion. There will be code in Pink which varies from implementation to implementation, but it must be carefully controlled. Note that "portable" does not mean "least common denominator". Every platform will have distinguishing features which Pink can and must take advantage of. Parts of the system may change significantly from platform to platform. Again, the key is management of assumptions to give the most flexibility possible.

Performance

Like any large system, Pink carries the risk of inadequate performance. The processor hasn't been designed which can't be brought to its knees by inattention to this goal. Pink must perform well on Mac II class machines, to the point where users will not perceive that they are losing something by running it.

Robustness

Anyone who uses the Macintosh today for any length of time (especially with INITs) must notice the general lack of reliability in our current systems. Applications crash randomly and non-reproducibly, documents and system files are corrupted, and disks even get trashed.

The problem is not the software, Apple's engineers, or the third party developers (well, maybe the third party developers a little). The problem is that the architecture itself is prone to these kinds of problems. It is very easy for a programmer to make a mistake which can cause subtle, difficult to find problems. The Macintosh human interface allows for the human frailty of its users, but the Macintosh architecture makes no allowance for the human frailty of engineers. Pink is more robust against the simple kinds of errors that engineers make. In addition, Pink takes greater care with data integrity so that such errors don't place the user's data at risk.

An example illustrates why architecture places severe constraints on our ability to achieve such goals. Any system which supports so-called unsafe languages (namely almost every language in wide use: C, Pascal, etc., but not Lisp or Smalltalk) has the problem of incorrect pointer references. For example, tired programmers (not enough Jolt — or too much?) attempt to dereference a pointer which contains zero. On Macintosh, this results in silently picking up (or worse, modifying) a value in low memory, an area which contains important system global variables. The program (or system) may run on for a considerable length of time before crashing. These global variables cannot be moved because their location is embedded in several thousand Macintosh applications: the architecture prevents a solution.

Pink solves this problem by eliminating low memory from the address space. The first N (currently 16 megabytes) locations of the address space simply don't exist, and any reference results in an immediate error. This is just one example of how an architecture can plan ahead for programmer errors.

Empower Developers

In addition to goals motivated by Apple's needs, Pink has the goal of making it easier for developers to build great applications. There are two thrusts to this goal: first, to allow the developer to concentrate on their application domain by removing much of the boilerplate code associated with developing a Macintosh application. Second, to raise the ante for applications by introducing major new system-wide features which enhance applications.

Saving a file is an excellent example of the first area. There are more than *twenty steps* to follow to correctly save a file in a way that is AppleShare friendly, deals with disk full, handles errors, works around Poor Man's Search Path, etc. These steps are documented in twenty different places. MacApp solves the prob-

lem by dealing with the issue itself, asking the developer only to provide the data to be saved. Pink takes the same approach.

Examples of the second area are linking, provided by CHER, and the Albert 2D/3D graphics system. CHER empowers developers first by providing lots of default functionality to every application, and second by allowing applications to build on top of what CHER provides. Albert empowers developers by giving them an extremely rich (but non-fattening) graphics system with more capabilities than anything else in the industry. Developers don't use transformation matrices, anti-aliasing, and 3D in their applications today because there would be no time left for the application after implementing all those capabilities. If it's built in, people use it.

Support the Interface

The Macintosh has a well defined human interface, and a set of human interface design principles, but almost no system support for implementing those principles. Consequently, developers must make a major effort to support those principles, and some don't get implemented at all. For example, direct manipulation is a cornerstone of the Macintosh philosophy, but very few applications implement it. Many Macintosh applications could exist comfortably on a system with a character display, as long as it supported windows, pull-down menus, and dialog boxes.

Pink directly supports the human interface. The path of least resistance for developers is to conform to the human interface exactly, with deviations requiring extra work. Pink also adds support for direct manipulation, so that it is easy for a wide range of applications to make use of it; in fact, it will be required to fully integrate an application into Pink at the desktop level.

A Global System

The Macintosh is one of the few computers that was designed from the first to work well around the world. Unfortunately, it aimed too low; it really only works well with European languages. By the time support for non-European languages was added, much of the application base was in existence; in addition, the problems caused by not including the Script Manager at the beginning have impeded developers from writing truly global applications.

Pink is a world wide system from the ground up. All system functions which deal with natural language are capable of supporting any language. Pink systems also support use of multiple natural languages at once.

Architectural Principles

This section discusses some techniques which are not goals themselves, but are intended to help us reach the goals discussed above.

All Interfaces through Objects

In order to get the most flexibility we can (by hiding as many assumptions as possible), the only client interface to services is through classes and objects of those classes. Among other things, this means that the following (usually important) concepts should *never* be part of an interface: messages, file formats, data formats, IPC. Note that these are all services themselves which can be used in the *implementation* of other services, but that they should never be part of that service's interface.

For example, you may well implement a service using the Scream client/server classes, but that fact should be completely hidden from the clients of that service. After all, you may want to change to an implementation based on shared memory and libraries instead of messages and servers, and the client doesn't need to care. Part of the problem with the Macintosh architecture is that interfaces are defined at too low a level ("the parameter block is laid out like this", "send a message which has these bytes in this

order", etc.).

Just using objects isn't enough; the objects must be designed correctly. Objects in Pink are defined in terms of the abstraction being presented to the client, not the implementation. It's quite easy to spill your implementation's guts through a class interface. The key is to design the class thinking about it from the *client's* point of view. What are the entities being dealt with? What do I need to know about them? What operations can I perform on them? These are the key object design questions.

Manage Commonality through Inheritance

Commonality in software systems has traditionally been managed by commonality of implementation. For example, UNIX[1] systems manage devices by making everything look like a block device or a character device. Device specific features are glued on through extensions. The Macintosh architecture uses the same approach: devices all support the same small set of calls, except that there are usually 1,897,422 variants of the control call to handle the specific attributes of a device.

Type inheritance provides a better way. A base class defines an abstraction and interface which is common to many objects. Specific objects derive from that base class, declaring themselves to be subtypes: they implement the common protocol, plus specific features unique to themselves. Several levels of abstract base classes can be defined, yielding successively more refined points of commonality. Also, objects can inherit from multiple base classes, thus supporting more than one shared protocol.

The benefit to clients is that they need only deal with the level of detail that is required by talking to the abstract base class and making themselves independent of the details of the subclasses. This allows those details to change, or new subclasses to be added, without breaking existing software.

Note that inheritance of code is a different matter. This has nothing to do with type relationships, and should be dealt with by has-a relationships (member objects), private base classes (a special kind of has-a relationship), or protected interfaces. Of course, it's OK (and common) to inherit both code and type from the same base class.

An example of these techniques applied to the driver world follows. An abstract class named TMassStorage defines the concept of a block storage device. Another class, MSCSIDevice, expresses the concept of a device on the SCSI bus. Finally, MNubusDevice models devices on the Macintosh NuBus. The file system only cares that something is a TMassStorage, not where it lives. A specific device might inherit from TMassStorage and MSCSIDevice: a typical SCSI storage device (in fact, this is so common another abstract base class is appropriate: TSCSIStorageDevice). On the other hand, it might be a high performance disk with a special NuBus interface card; its driver would inherit from TMassStorage and MNubusDevice. In each case, clients deal only with the protocol they care about, and unnecessary detail is hidden. Similarly, a video NuBus card would be a TVideoDevice and an MNubusDevice.

Leverage Where Possible

Using an existing object rather than inventing a new one is a good way to achieve several Pink goals. Less code means a smaller memory footprint, which yields better performance. Fewer classes means less for the developer to learn. Less to implement means fewer bugs, leading to a more robust system. It also means we get done faster, leading to bigger profit sharing checks (this is an implicit Pink goal not mentioned above).

For example, Pink has a set of collection classes which implement common data structures from Computer Science (note capital letters), such as stacks, sets, trees, etc. Pink uses these classes heavily to avoid reinventing the wheel. Similarly, the many building block classes Pink provides allow the developer to concentrate on his or her application rather than reinventing common tools.

---

1. I don't care whose trademark it is. So there.

Sometimes this requires battling habits learned in previous lives. For example, the AppleTalk datagram protocol (DDP) has a concept of a socket. A machine may have up to 256 sockets, which are mostly allocated dynamically. The first inclination might be to use a bit vector to keep track of which socket numbers have been allocated. This leads to a fair amount of custom code. If you look more closely at the application, however, you notice that first, there is only one such data structure per CPU, and second, performance is not a critical issue. Instead of building a custom data structure, the existing classes TSet and TCollectibleLong could be used to implement a set of integers, which is precisely what a bit vector is.

Naturally, leverage does not mean "one size fits all" (TProcrustes?). In another context, speed or space performance may demand that a custom data structure be used. That's OK. As Einstein said, "Everything should be as simple as possible, but no simpler."

Frameworks Protect Subsystems from Each Other

Use of objects and inheritance help isolate clients from assumptions and unneeded details, but what about developers implementing derived classes? Also, use of objects isolates the developer from assumptions about the objects, but not from assumptions of structure. Both of these problems can be solved by the idea of frameworks.

The Lisa Toolkit and MacApp were the first object oriented frameworks. MacApp defines several base classes, among them TApplication, TDocument, and TCommand. These classes hide the details of the *client* from the *system* at the same time as they hide the details of the system from the client; the information hiding goes both ways. Parts of the classes' interfaces are used by the MacApp system, and parts are used by developers. The relationship between these parts is handled by the class, thus isolating the developer and the system from each other. MacApp deals with concepts which are important to it, and the developer deals with concepts which are important to him or her.

Let's recycle an earlier example, saving documents. MacApp's document framework class, TDocument, defines two methods in its protocol: TDocument.Save and TDocument.DoWrite. MacApp itself calls TDocument.Save in response to the Save command from the File menu; the TDocument.Save method represents the abstraction of a File menu Save command. The developer overrides the TDocument.DoWrite method when creating his or her own subclass of TDocument, and makes it write out the contents of the document. This is only one of the twenty or so steps mentioned previously, but it is the only one the developer cares about or which varies from application to application. The rest of the document saving protocol is implemented within the TDocument base class. Should one of the other steps change, or more steps be added, the developer is protected from it.

The MacApp application framework is thus protected from the details of specific documents, and the developer is protected from the details of documents which are irrelevant. The TDocument framework class has interfaces which go both ways: one for clients, and another for subclasses (of course, these can overlap somewhat).

Pink uses frameworks extensively. In addition to the MacApp application and document frameworks, Pink adds: a client/server framework, a graphics device framework, a concurrency control and recovery framework, several frameworks for implementing different kinds of device drivers (NuBus, SCSI, video, disk, etc.), a text editing framework, a framework for file systems, and more. This structuring technique made MacApp possible, and it's an important part of Pink.

Let Resources Find You

Traditionally, programs have names of resources or collections of resources hard wired into them, and go out looking for these resources. Yet programs do not usually need to know this information, and frequently it's only used to put up a list for the user to choose from. Needless to say, having many different programs write their own code to find things and then to put them up in lists for the user to choose is not in keeping with the Pink principle of hiding assumptions and information.

Pink instead takes the approach that code should wait for resources to find it, rather than looking for resources. A form of this happens today on the Macintosh. The file system does not know about all possible devices which can act as disks, and go out looking for them. If it did, it would be impossible to create things like tape disks or NuBus disks, simply because the file system would have contained the (unnecessary) assumption that, say, only SCSI disks can contain file systems. Instead, the Macintosh file system waits for resources to find *it* through the drive queue data structure. Anything can declare itself to be capable of holding a file system by inserting an entry in the drive queue.

Pink takes this idea as an architectural principle, and takes it one step further. First, resources should register themselves with services; services should not go out looking for resources. This is a "bottom up" approach rather than the traditional "top down" approach. Second, whenever possible resources should register themselves on the desktop, and services should be told what resources to use via choices from the desktop, so that users have to remember only one way to "choose" things.

Here is one example of these principles in action. At boot time, the SCSI software must enumerate the devices attached to the system, and create device drivers for each device. However, this is the last time the SCSI devices need to be enumerated (except for diagnostic purposes, or if live attach and detach are allowed). From this point on the driver, *not* the SCSI software, is responsible for providing access to the resources it represents.

A SCSI disk would register itself on the desktop as a raw device, to allow the user to select it for formatting or partitioning. If it contained volumes, it would also register itself with the file system, to allow the volumes to be mounted. The file system would in turn register the new volumes on the desktop to allow the user to select them or open them. Note that in each case the services need only know the minimum information. The file system need only know that something is a storage device, not whether it is a disk, a tape, SCSI, NuBus, or whatever. The desktop need only know that something is a resource; when that resource is selected (by the user for, say, formatting) the application need only know it is a storage device. Similar techniques work for other expansion busses, like NuBus, and device types, like video cards.

This architecture solves several problems. First, by isolating applications from irrelevant information, it increases flexibility. For example, the Apple scanner application can only use the first scanner it finds on the SCSI chain. A Pink scanning application would use any scanner the user selected from the desktop, whether it was on SCSI, Nubus, or whatever (as long as it was the right "type" of resource).

Second, since resources are registered with the desktop, users have only one paradigm for using them. This prevents the confusion we have today, where SCSI disks are found in one place and selected one way, network resources are found in another place and selected another way, video resources are found in a third place and selected a third way, and so on.

Naturally, some things don't fit this model. Although fonts need to be manipulated on the desktop, I doubt users would like to select them this way when writing a document. Thus, some resources may need to be registered in more than one place and presented in more than one way.

Architectural Overview and Issues

This section gives a broad overview of the architecture of the system. It's supposed to be a complete synopsis, but will probably only approach that goal asymptotically. As I discuss each functional area, I will also enumerate the open issues and coming attractions.

The Opus/2 Kernel

Any view of the Pink architecture must start with the fundamentals of the programming model and environment. The foundation of that environment is the Opus/2 Kernel.

Opus is an operating system kernel which provides a small set of powerful abstractions. It follows in the same spirit of previous such kernels, like Stanford University's V and Carnegie Mellon University's Mach. The design goal of Opus is to abstract away the variations of machine architecture while at the same time remaining small, leaving as much as possible to code running outside the kernel.

The primary abstractions that Opus provides are:

- Multiple, independent virtual address spaces. Each address space is associated with a team (see below).
- Multiple mapped segments[2] within an address space. Each segment has a corresponding backing store, which may be a file on disk, part of the physical address space, or anything else. Segments are usually demand paged, unless otherwise requested. Segments can be mapped into more than one virtual address space simultaneously, allowing communication via shared memory.
- Multiple threads of execution within each address space. Each thread is known as a task, and the collection of tasks in an address space is a team (each team has at least one task). Tasks on a team share the entire address space of the team. All tasks execute in the "user" (unprivileged) mode of the processor Pink is running on. Tasks share the processor(s) on a machine via a preemptive, priority based scheduling system. Each task has its own stack.
- Message based communications between tasks. This IPC[3] mechanism allows tasks to communicate with each other synchronously or asynchronously, passing data between themselves.
- A facility for loading interrupt handling code into the kernel, and communicating with those handlers. On most processors (read: all the ones we know of today), these handlers run in the same processor mode as the kernel, usually the "supervisor" or "privileged" mode.

A set of classes provides the interface to these kernel abstractions; objects of these classes control the corresponding kernel resources. The entire Pink system is built on these abstractions. Except for interrupt handlers and the kernel itself, all Pink code is executed by one or more tasks running in the address space of one or more teams. Shared memory and IPC tie these tasks together.

In Pink, there is no distinction between system code and user code. Applications run as teams, but system services and I/O drivers run as teams (or in teams) as well. Thus, all services which are available to applications are also available to system code (obviously excepting circular dependencies, e.g. parts of the disk driver can't be paged).

Opus Issues

- Currently, access to shared memory is synchronized via semaphores implemented in library code. This function is likely to migrate into the Opus kernel, and may be replaced by a model of monitors and conditions.
- Certain real time applications such as sound, video, and human interface do not work well with a priority based scheduler. The system scheduling model is likely to expand to support real time requirements, by including concepts like deadlines or periodic scheduling (this may be in the kernel or layered on top).
- Currently, segments which do not correspond to physical address space must be backed by disk files. This will be changed to support segments backed by other stores, as well as segments (such as caches) which do not have backing store at all. On a related note, the interface for segments is currently closely tied to the file system. It will become more independent in the future.

2. OK, so it's a horrid name. So sue me.
3. An archaic acronym, for Inter Process Communication.

- Currently, IPC, semaphores, and paging do not take priorities into account (i.e., the queuing is "fair"). This is probably not appropriate for an interactive, real time system. We will experiment with these issues to find the right approach.

- Several clients need "copy on write" functionality for segments. This will be provided by a "lazy evaluation" copy function.

The Run Time Environment

The Pink run time system builds a programming model on top of the abstractions of the Opus/2 kernel. This programming model is designed to support the classic Algol family of languages: C, Pascal, Fortran, and so on. In addition, the run time supports an object programming model based on the semantics of C++. Finally, there is a set of features specific to Pink.

Shared Libraries and Classes

One of the most important run time abstractions is shared libraries. A shared library is a collection of functions and global data which can be loaded into a particular team. The code is shared between all teams which reference the same shared library; each team gets its own copy of the library's global variables. Shared libraries can refer to each other, and such references are resolved when a library is loaded. This may cause other libraries to be loaded in turn. All Pink system classes are provided via shared libraries supplied by Apple.

This does not mean that only Apple writes shared libraries. Indeed, the architecture allows an indefinite number of shared libraries, and allows third parties to supply them as well. In effect, anyone can write "system software". Shared libraries are like Macintosh toolbox traps, except they use normal language run time mechanisms, no one has to allocate trap numbers, and anyone can write them.

Shared libraries are normally mapped from disk files, but can also come from special disk partitions or ROM. Unlike the Macintosh, there is no difference between shared libraries loaded from ROM or disk. To support ROM, shared libraries can be patched on a function by function basis, regardless of whether a function is exported from the library.

In addition to the normal static references which are resolved at library load time, Pink allows libraries, classes, and functions to be loaded at any time on program request. A program can request that a library be loaded by name, and can then instantiate an object by class name, or look up a function by name. This capability greatly increases the flexibility and expandability of the system, and enhances the power of inheritance.

Language Support

The Pink run time also has the following features:

- Standard C, C++ (minus AT&T's task package), and SANE (including complex) libraries.

- Fast semaphores for synchronization of shared memory.

- Very fast storage allocation which is safe for use by multiple tasks. This is *not* a relocating storage allocator like the Macintosh; allocated blocks don't move unless requested.

- Support for a software exception handling model. This model is based on termination semantics (like CLU, Ada, and MacApp), and will support the semantics being developed for C++. All unusual conditions in Pink are expressed through this mechanism rather than through error codes.

- Support for debuggers and handling of hardware (processor) exceptions.

Run Time Issues

- The load files used for shared library files are currently Blue application (resource) files. These will be replaced by a native load file format when the Compiler Technology project (CompTech) compiler becomes available (this compiler is being built by the Class group).

- The design for how shared libraries are located is not complete. There will be "published" shared libraries for everyone to use, but how library searching is handled when a developer doesn't want to globally publish one is not clear yet.

- The virtual function tables used for virtual function calls are currently constructed at compile time and live in the global data for a library. This means 1) new virtual functions cannot be added without recompiling clients, and 2) virtual function tables are not shared between teams. When the CompTech compiler is available, virtual tables will be constructed dynamically by the run time system, alleviating both these problems.

- As mentioned in the kernel section, semaphores will probably become a kernel function.

- The exception handling mechanism is not nearly in its final form. It will mutate in that direction over time, and will be in final form when the CompTech compiler is available.

The Foundation Classes

Pink provides a large set of classes of general utility to programmers. These are used widely throughout the system. They include:

- A set of collection classes which implement standard data structures (Utility Classes). This includes a set of high level classes (Bags, Sets, Queues, Stacks, Dictionaries, etc.), as well as low level constructs for those needing finer control (hash tables, red-black trees, etc.). In addition, other common data structures are provided, such as directed and undirected graphs, and a class for time stamp concurrency control.

- An abstract class which represents a stream of bytes, and a framework for flattening objects onto such streams and restoring them from such streams (Cheetah).

- Numerical classes which support 64 bit integers and fixed point numbers. In addition, SANE may be extended with new numerical types, such as higher or arbitrary precision numbers or interval arithmetic.

- A framework for concurrency control and recovery (Credence). These classes allow shared data to be updated in a way which guarantees its integrity. They can be used to prevent data from becoming corrupted due to software crashes or some hardware crashes. These will be used by the Pink file system and other components; Pink will have no "Disk First Aid" like applications.

- A class which allows objects to be stored in files and retrieved by keys, which may themselves be objects (PsychoKiller).

- A class which allows short unique identifiers to be assigned to entities in an extensible fashion (Tokens).

- A set of classes for implementing client/server relationships (Scream). This is widely used throughout the system to implement classes which do their work by communicating with servers.

Foundation Issues

- It is not clear there are resources to implement any of the proposed SANE extensions.

"Operating System" Services

Many of the services that traditionally would be provided by an operating system are instead implemented via shared libraries and server teams under Pink. These services include:

- A set of framework classes and servers for implementing device drivers. These include frameworks for SCSI, Nubus (or other expansion schemes), mass storage devices, video, serial devices, and ADB. In each case, new kinds of devices can be added by deriving new classes from one or more of the base classes supplied by Apple. In many cases, developers will not need to write their own interrupt code but can instead use that provided by the framework class.

- A set of client classes, framework classes, and servers for implementing file systems (Pluto). These will provide a common interface to multiple kinds of file systems, including Pink's native system, HFS/MFS, MS-DOS, CD-ROM, and so on. New file systems can be added by deriving new classes from the file system framework classes. The Pink native file system will support new features such as large volumes, full international file names, arbitrary file attributes, access control for local volumes (as opposed to only AppleShare volumes), and file groups (a generalization of the Macintosh data and resource forks).

- A mechanism for indexed access to system resources (Rainbow Warrior). This is for use in situations discussed above (like Fonts) where desktop access is not enough.

- An event server for coordinating events from input devices and distributing them to applications. It also enforces the fundamentals of the user interface to prevent errant applications from locking up the system.

- A layer server which arbitrates screen real estate among all applications. It doles out space on the screen and synchronizes drawing with screen rearrangement so that application drawing remains within the appropriate areas.

"OS" Issues

- It is not clear yet whether all individual devices will be controlled by separate teams or whether some devices may be controlled by objects operating within shared teams.

- The functions provided by Rainbow Warrior overlap somewhat with those provided by the Desktop; they need to be coordinated.

- There is a major missing component, namely security and authentication services. This is necessary to guarantee the security of the local file system. Access to the file system, to certain hardware resources (such as mass storage devices), and to certain OS features (loading ISRs, creating physical segments) must be authenticated if the local file system is to have any more security than it does today. A recent survey stated that around sixty percent of all Macintoshes were used by two or more people, so this seems like a good idea. This issue must be resolved soon. A related issue is how to deal with people as objects in the system; this is needed for collaboration as well as authentication.

- Another area which we must consider is system reliability. If Pink systems will be used as servers, we may need some or all of the following features: storing information redundantly to protect against media failures (including disk mirroring), logging of soft and hard errors to spot potential problems,

some measure of fault recovery. Do we want to do these? Also for servers, we may want to allow volumes to span multiple physical disks.

- Currently there is no strategy for dealing with power management, an important issue when running on future Pink-capable battery-powered machines. The key requirement is to be able to tell that the machine is "idle", i.e., no useful work is going on, so that it may be put to sleep. Since many teams and tasks will be running even in the idle state, we need to put a framework in place for extracting this information.

- The system boot sequence has not been designed yet. Pink may or may not be in ROM, but we will probably design a boot sequence capable of dealing with either eventuality (if only to aid during development) and of booting over the network. This work will commence shortly.

- Rainbow Warrior provides a change notification facility, as do CHER and some other components. These need to be unified, or at least be expressed using a common set of base classes.

The Graphics System

The Albert graphics system provides comprehensive 2D and 3D rendering and modeling capabilities for Pink. It surpasses most of the combined functionality of Color QuickDraw, PostScript™, and Renderman™. Here is a list of Albert's capabilities:

- Device and resolution independence.

- Complete 2D (3 × 3) and 3D (4 × 4) transformations. All geometry and transformations are based on floating point coordinates.

- A large set of 2D primitives: lines, arcs, polylines, curves (beziers and non-uniform rational B-splines), rectangles, round rectangles, polygons, and ellipses; these all include hit testing. 2D geometric collections: paths (open) and areas (closed). Sampled images (pixmaps).

- Rendering of text integrated with the rest of the 2D primitives. Character glyphs can be part of a geometric area. Fonts and glyph rendering are done with objects, so multiple font formats and rendering techniques can be supported: initially, True Type (Bass) and bitmaps.

- An extensible set of 2D rendering techniques including: inset, centered, and outset pens; line thicknesses, joins, and end cap styles; filling and/or framing. New kinds of rendering techniques can be added by developers.

- A large set of 3D primitives: lines, polylines, and curves; boxes, polygons, meshes, polynets, boxes, swept and extruded shapes, and 3D spline surfaces. 3D geometric collections: paths and surfaces.

- An extensible 3D rendering and viewing model that includes camera and light source modeling and many shading techniques. New kinds of rendering techniques can be added by developers, similar to programmable shaders in Renderman.

- An extensible 2D modeling framework which can tag objects with rendering attributes and transformations, and includes the ability to create nested groups.

- An extensible 3D modeling framework which includes support for sweep and extrusion.

- A graphics output device model which is extensible to support many different rendering techniques, including standard frame buffers, graphics accelerators, and spooling of graphics for printing.

Albert implements all these capabilities with a small set of abstract classes, among them MGraphic (the abstract class for modeling), TGrafBundle (object rendering attributes), several geometric classes, TGrafDevice (an abstract output device), TGrafPort (graphics context for rendering), and TMatrix and TMatrix3D (transformation matrices).

Albert Issues

- It is not clear how to handle developer supplied extended rendering classes (e.g., for shading) when sending graphics output to a graphics accelerator or other external rendering engine, or when putting graphic objects into a document which may be transferred to another system.

- Still to be defined is the architecture for handling buffering, compositing, sprites, and animation. This is needed to support the mouse cursor, and dragging of objects in front of other, animating objects. It would also speed window dragging and handling of updates by obviating the need for redrawing windows in all cases.

- The architecture for text and fonts is still being defined. Font raster caching must still be designed.

- The architecture for dealing with color spaces and CLUT output devices is yet to be defined.

Application Support

Pink contains several components which are directly concerned with supporting applications. These are the Application framework, which supports graphical user interfaces, the Document framework, which supports the Pink document model, and the scripting framework, which ties applications into the system script facility. These are described below.

The Application Framework

The Pink application framework (an evolution of the one in MacApp 2.0) provides an environment for supporting graphical user interfaces. Specifically, it includes:

- A model of input devices and events which can be extended to support new kinds of devices, such as data gloves or 3D input, and an open ended set of events.

- An event distribution model which routes events to user interface components and allows those components to be hooked together, without explicit action on the programmer's part.

- A view system which partitions the display of an application into self-contained components. Views are similar to windows, except that they form a hierarchy rather than a simple collection. They also provide a framework for managing interaction.

- A set of user interface components (windows, buttons, menus, scroll bars, etc.) built on the foundation and extensible by third parties. This also includes views for common presentation techniques such as tables of items and collections of graphics.

- A framework for tracking user interface actions which includes standard support for tracking the mouse, for handling double clicks, and for dragging elements within views, between views, and between windows. The dragging framework also provides a means for a dragged element to communicate with other elements it is dragged over.

The two primary building blocks of the application framework are the *responder*, which is an object that can receive events, and the *view*, which is an object that can contain graphics and receive positionally directed events (such as from a mouse). Views are also responders; they are a subclass.

Events are usually routed to responders in three ways. Non-positional events, such as keystrokes, are sent to a specific responder associated with the current frontmost layer; this responder is called the *target*. Positional events, such as mouse button down, are routed through the view hierarchy to the view which was under the mouse when the event occurred. Finally, events can be routed directly to a specified responder.

In addition to receiving events through the event distribution mechanism, responders can receive input directly from other responders. This is done through *output ports*, which connect output from one responder to the input of another. Thus, responders can be hooked together in different configurations without having to write code to do so. This is essential to a user interface construction capability, like that being planned for the Hoops development environment (Pink's native development environment, also being built by the Class group).

Views can be thought of as virtual paper, and the view hierarchy as a mechanism for arranging those papers to make a coherent interface. Each view has, at a minimum, a container (the view within which it appears when displayed), a position within that container, a coordinate system (implemented by a transformation relative to its container's coordinate system), and a boundary (which may be a rectangle or an arbitrary Albert area). Views also know how to display themselves when asked (much like an Albert MGraphic). When displayed, they clip to their boundaries, any ancestor's boundaries, and are hidden by views in front of them.

The tracking framework is based on the *tracker* object, which represents an interaction in progress. The tracker encapsulates all knowledge of the interaction. Standard subclasses of tracker provide support for mouse tracking and dragging.

The Document Framework

The Pink document framework (CHER) provides the abstractions and frameworks which implement the standard Pink document model. In particular, it provides for:

- Collaboration: every Pink document built using CHER supports both simultaneous and sequential collaboration between users on that document. Users can operate on the document at the same time, either on one machine or across a network, and see changes made by others reflected immediately. Annotations allows users to communicate with each other when separated by time.

- Hypertext: CHER documents can contain links to other CHER documents. These links are bidirectional and can be used purely for navigation (to find and open the other end) or to push or pull data (allowing a "hot link" capability). Developers can use the underlying link technology to also provide their own custom capabilities.

- Multi-level undo: like MacApp, CHER provides a framework for implementing undo. Unlike MacApp, CHER's undo is multi-level; users can undo back as far as system resources permit. User actions are logged using Pink's recovery features, so users need not save to preserve their work from crashes.

- Content based retrieval: CHER provides a framework for indexing and content searching, to allow documents to be located by content.

CHER is at the heart of Pink applications because Pink has a document-centered user model: users deal with documents, not with applications. CHER provides a subclass of the application framework which knows how to deal with documents and knows how to integrate with the desktop. Applications are invoked whenever necessary to perform services in a manner transparent to the end user.

The Scripting System

Pink provides a system which allows end users to automate repetitive tasks. It has the following features:

- Scripts can be recorded, and the system watches to try to discern repetitive actions. The system uses these to try to generalize the actions the user has just performed.
- Scripts are represented in a visual fashion similar to storyboards or film clips, to make them more accessible than textual languages.
- Scripts are universal; the scripting framework is integrated with the document framework, making it easy for all actions taken by the user to be recorded and played back.

Application Support Issues

- Much of the view system still needs to be fleshed out. There are synchronization issues which need to be resolved, additional support needed for processing of updates by other threads (this may become the default), support needed for off screen buffering of views, and Albert support for sprites must be integrated when available. Palette management for support of CLUT devices awaits Albert support. Support must be added for tracking view sizes and resizing views. We may want to add some kind of framework for automatic view layout, to ease the burden of localizing applications. Something like the "boxes and glue" model of Interviews might work.
- The event handling system still needs to resolve some synchronization issues, and the dragging framework has not yet been designed. Most of the standard human interface components have not been built yet, and the human interface itself is still being designed.
- The synchronization issues between the event handling and tracking framework, background view updating, and documents have not been resolved.
- There is another kind of collaboration beyond that provided by CHER: collaboration at the desktop (as opposed to document) level. This is similar to what Timbuktu and other applications do on Macintosh today (screen sharing). How this should be done in Pink is under investigation.
- Classes for exporting and importing foreign file formats (Blue and others) are needed but have not yet been designed.
- Help support will be integral to the application framework, but we are waiting for a human interface design before designing the software.
- Work on the scripting system is just underway, and many issues remain to be resolved.

Text and International

Initially, support for text on the Macintosh consisted of TextEdit (32K characters, plain text) and the International utilities (Roman languages). Since then, the Script Manager has been added to support non-Roman languages, and TextEdit has been extended to support multiple fonts and styles. Unfortunately, because there is no support for sophisticated text editing and because the Script Manager is not integrated, every developer who wishes to support text must write their own editor, and most skimp on non-essential features. Very few use the Script Manager because of the effort involved, and many do not implement all of the Macintosh human interface (the word processor I'm using to write this does not support "intelligent" cut and paste).

Pink provides extensive support for sophisticated text editing. It includes a text editing framework which supports arbitrary length text and an extensible set of styles. The text system supports sophisticated typographic controls, text flow across blocks, high quality line breaking, and more. The framework is intended to provide a highly functional base level of text support in Pink, as well as a toolkit for construction of high end text applications (word processors and page layout programs). The goal is to prevent developers from having to reinvent the wheel so they can concentrate on providing distinguishing features. The text framework can be overridden to provide such features (like footnotes or indices).

The text framework works closely with the international classes so that all text manipulation is fully international by default. Pink uses a 16 bit character set (Unicode) and is capable of handling multiple scripts with different writing directions and input techniques. Because of the integration between text and international software, input of complex scripts such as Japanese can be done inline while editing. Pink text can take advantage of such sophisticated options as automatic kerning, ligatures, contextual forms, and optical alignment. Pink will support simultaneous use of multiple scripts within documents, and will be able to switch the "system" language (used for menus and controls) while running.

In addition to text editing, layout, and formatting, Pink comes with utilities to allow developers to build worldwide applications. Calendar calculations, date input, date output, numeric formatting, text parsing, and text searching can all be done in a fully international way. Users will also have much more control over configuration; they will be able to choose from multiple languages, collation rules, local rules (such as time zones), number formats, and so on.

Text and International Issues

- Implementation is just beginning on the international classes.
- There are no Bass fonts with the extra information needed to test some of the text layout capabilities. Indeed, there are no Bass Unicode fonts.
- There are some problems to be resolved with allowing developers to add styles (one of which is the same issue as above for Albert custom bundles: how do you transmit such data to a different machine?).
- There is no architecture yet for linguistic services like dictionaries and thesauri. In addition to the benefit to end users and applications, such services can be extremely useful to content based retrieval mechanisms like CHER. A dictionary or thesaurus can be used to prevent queries from being interpreted too literally, for example by stripping out irrelevant word endings. A thesaurus might even be used to let a query for "car" find "automobile", for example.

Printing

The Pink printing system is composed of several pieces which work together to provide a flexible and extensible system. These are:

- A framework for objects which can be printed, which helps them deal with pagination, page size, and other printer attributes. This framework has a set of useful defaults which allow most applications to print with very little work.
- A set of classes for characterizing printer attributes and print jobs. This includes user interface classes for presenting a standard human interface to select such attributes.
- A framework for building print drivers, which lets developers define new kinds of printers with new capabilities. This is similar to the application, document, and other frameworks in Pink: developers need only define the unique behavior of their particular printer.

- A spooling architecture which will do all printing in the background.
- An architecture for dealing with scanners. This consists of a set of classes for use with CHER to allow images to be input into applications, plus a desktop scanner object (a driver).

Printing Issues

- We would like to support multimedia output (sound, animation, video) in the printing architecture. This is still under investigation.

Time, Sound, and Animation

Pink has a set of classes dedicated to timing and time-dependent media. These include:

- Classes for getting the current Julian date and time. Pink uses Universal Time (a.k.a. Greenwich Mean Time) internally. Calendar calculations and date input and output are handled by the International software.
- A set of classes for timing and alarms. The most important class is TClock, which represents a source of timing information. Different clocks can run at different rates; clocks need not advance in real time. TTime objects represent timing information relative to specific clocks, and TAlarms can be used to perform actions at a given time or periodically.
- A set of classes for dealing with sound and speech. These classes represent audio sources, processors, and destinations, based on physical metaphors (e.g., sampled sounds, mixers, microphones, speakers, telephones). These audio components are patched together much as real sound equipment would be. Third party developers can create new kinds of sound objects, and sound objects can be composites of other sound objects.
- A set of classes for dealing with time-sequenced data, such as MIDI information.

Issues

- The use of UT internally implies that we must be more determined about finding out the current time zone. Also, we could take advantage of network time services (including existing ones such as NTP), but no one is looking at this.
- There are multiple system components that use the metaphor of hooking things up: the user interface (responders and output ports), the sound classes, and the time sequenced data classes. The common ground needs to be determined (if there is any) and shared as a set of abstract base classes.
- The animation and video architecture has not been defined yet.

Networking and Communications

The Pink networking classes provide:

- A framework for implementing a wide variety of networking protocols. In some cases, these protocols can be mixed and matched.
- Support for simultaneous connection to multiple networks ("multi-home").
- A class which represents a network resource, which can also instantiate transaction or streaming connections to that resource, independent of the network protocols used.

- A networking and communication component to the Blue Adapter, which emulates native Blue networking services by calls on Pink network services.

- A set of classes for accessing the capabilities of MCP cards which utilize the A/Rose kernel, to enable use of the many NuBus cards which use the MCP platform.

N&C Issues

- There is currently no equivalent to the Communications Toolbox defined. Something on the order of a TSerialConnection desktop object seems necessary to allow configuration of native Pink software using serial connections.

- There are human interface problems involved in mix and match protocols and multiple networks.

The Desktop and Finder

In order to meet its goal of extending direct manipulation and physical metaphor into every last nook and cranny, Pink assigns the desktop itself a much more global role than is true on Blue. The Pink desktop is not just a window onto the file system; it is the basis for the presentation of everything that is going on in the "world" inside the computer. Desktop objects represent physical devices (such as expansion cards and ports), networks resources, disk drives (not just volumes), and others.

Pink also separates the user interface portion of the desktop (Thor, the equivalent to the Macintosh Finder) from the underlying abstractions, which are available through a set of classes (Odin). Objects on the desktop map directly into Odin objects. Odin itself only deals with the abstractions which apply to all desktop objects: containers, moving, renaming, copying, opening. It is really a shell for implementing and using desktop objects.

What desktop operations mean for individual desktop objects is defined by the objects themselves, by deriving classes from the abstract base classes Odin provides. Naturally, Apple will provide a rich collection of such concrete classes which can be used to implement many common objects (disk drives, files, volumes, folders, NuBus cards, etc.). Also naturally, third parties can add their own meanings by subclassing any of the classes Apple supplies.

In keeping with the Pink principle of letting resources find you, most programmers will deal with the desktop by accepting resources specified by the user. Such resources can be supplied implicitly (e.g., an application is started when the user clicks on a document, and is handed an object representing that document), or explicitly (e.g., the user drags a printer into a "socket" in a window to set the default printer for a document). The connection between desktop objects and potential users of those objects is made using the dragging and type negotiation protocol defined in the Pink application framework: a target can accept a desktop object if the protocol determines that they have a "language" (a type, representing a protocol) in common.

Developers will also deal with the desktop by creating desktop objects of their own: the example given in the principles section was of a TMassStorage object creating a desktop object to represent the raw device, and of the File System creating a desktop object to represent the volume or volumes on such a device. Because objects find the desktop, rather than *vice versa*, the mapping of resources into desktop objects can vary widely across system components and over time.

In order to increase the consistency of the metaphor, Pink discards some components of the Blue interface. Like Lisa, Pink uses a document metaphor; applications are only resources that need be present to enable certain kinds of documents to be opened and manipulated. The user opens a document, and does not (consciously) "run" an application. Similarly, Pink discards some File menu commands: New, standard file, Save, and Quit.

New documents are made by copying stationery (the application itself could serve as default stationery to avoid the Lisa problem of throwing all your stationery away). Documents are saved automatically; reversion is handled by multiple level undo and "snapshots" of document versions. Standard File functions are handled by the Finder; users will not miss Standard File as long as all the shortcuts (and speed) it provides are in the Finder too. Quit is superfluous since users work *on* documents, not *in* applications. Each document will appear separately, rather than all documents of a given type being lumped together if more than one is open. When and how application code runs is determined by the system, not the user.

Finally, the Pink desktop will reduce the distinction between desktop objects and data. Desktop objects can be dragged into containers, and in Pink selections can be dragged between documents. We would also like to be able to drag selections into containers (perhaps creating new documents) and desktop objects into windows and documents (as in the printer selection example above).

Desktop Issues

- Ideally, file system operations should go through the desktop to ensure consistency. This may be unacceptable from a performance standpoint, in which case a scheme would be necessary for the file system to notify the desktop (or other clients) about changes.

- The handy-dandy Acme Dragging and Type Negotiation Protocol™ has not been designed yet (although the type negotiation part has been implemented for CHER).

- One advantage of the multiple metaphors in today's Macintosh is that they can be tuned for experienced users. Navigating Standard File or the Chooser can be very fast. Sometimes the needs of experienced users lead to metaphors being bypassed altogether: many users use Switch-A-Roo to change bit depths on video devices simply to avoid having to get out the Control Panel, scroll to the Monitors cdev, click on it, click on the monitor, and set the depth. If we are to use physical metaphors pervasively as our user interface, the needs of experienced users for "fast paths" cannot be ignored. The integration of the Standard File keyboard shortcuts into Finder 7.0 is an example of how this can be done.

- The design of the desktop classes depends on the Pink metaphors, and thus is dependent on the design of the general Pink human interface. The latter is not nearly completed.

- Desktop objects other than the ones supplied by Apple cannot be copied to other systems unless the (custom) implementation goes along with the object. This applies to objects on removable media as well. This is the same problem faced by other Pink components.

- The pervasiveness of the desktop means that it will permeate every corner of Pink, and thus affect the design of many components of Pink. However, the desktop classes are only now being designed and implemented.

- Once the desktop classes are available, there are an awful lot of desktop objects and utilties which need to be designed and implemented. All functions that are currently handled by Chooser, Control Panel, HD SC Setup, Font/DA Mover, *ad nauseum*, must be built. This is a lot of work.

Adapters

Pink is designed to allow foreign operating environments to be emulated concurrently with native Pink applications. These emulators are known as adapters. There are two currently being planned:

- The Blue adapter will allow Pink to run existing Macintosh applications and some subset of other Blue software (probably DAs, cdevs, and some INITs). The technology used will be very similar to that in A/UX 2.0; the major difference is that unlike A/UX, Pink has a native video architecture and cannot use the Blue model unchanged. In addition, some measure of data exchange between the two worlds

will be possible: at least clipboard and file import/export, and possibly some connection with CHER.

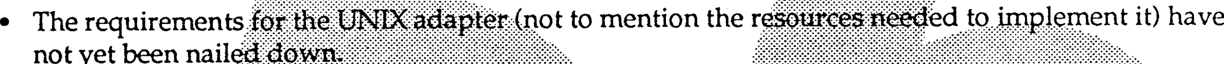
- The UNIX adapter will allow some subset of applications written for UNIX to run under Pink. The set of applications which run depends on what UNIX applications, and thus what UNIX features, we consider important to emulate. The goal is to let applications which use standard System V, BSD, or POSIX interfaces to run, and also to support such common UNIX accouterments as X. It may even be possible to support one of the application binary interface (ABI) UNIX standards and thus run existing binaries, but given the projected installed base of Pink this may not be necessary.

Adapter Issues

- The technique for handling graphics in the Blue adapter has not been decided. Allowing QuickDraw to continue to draw to the screen guarantees pixel perfection, and the Layer Server was designed to allow multiple rendering systems. However, this doesn't work well on devices which are not a multiple of 72 DPI, and it breaks down completely for graphics accelerators. Under Pink, a graphics accelerator card will be running Pink code, not Blue. It also doesn't work for devices with non-scaling transforms (like rotation, for example).

- How much software besides applications can run in the Blue adapter is an open question. DAs, certainly, probably some INITs and cdevs. Drivers are iffy. It may be possible to wrap up Blue video and disk drivers (by far the most important) to make them run under Pink, but this is not guaranteed by any means. Rewriting drivers is a pain, but the Pink I/O frameworks should make this much easier.

- The requirements for the UNIX adapter (not to mention the resources needed to implement it) have not yet been nailed down.



56

Pink Human Interface

56

Pink Human Interface

Making the computer a little simpler

- **Simplifies what is**
- **Adds what's new**
- **Restores the fun**

Lee Honigberg
Frank Ludolph
Annette Wagner

56

Table Of Contents

Introduction 4

Extending the Macintosh Design Principles 5
- Metaphors from the real world 5
- It works the way you do 5
- Direct Manipulation 6
- See-and-Point (instead of remember-and-type) 6
- Feedback and Dialog 6
- User Control 7
- Safety (a.k.a. Forgiveness) 7
- Consistency 7
- Perceived Stability 7
- WYSIWYG 8
- Aesthetic Integrity 8
- Fun to Use 9

A Pink Interface Proposal 10
- Macintosh Abridged 10
- The New Layering Model 11
- The New Document/Application Scheme 12
- Multitasking 14
- The Network 15
- Direct Manipulation Cut, Copy, and Paste 17
- Linking 18
- Scripting 18
- Post-its 18
- Filling Out Forms 18
- Groups of People 22
- Collaboration 22
- Project Lists 23
- Print Shop 23
- All the Other Stuff Now Found in the System Folder 23
- New Cool Things That Are Small or Less Than Pervasive 24
- Quiz 26

The Architecture of the User Interface 28
- Object 28
- Operations 30
- Interaction 31
- Taxonomy 33

56

Introduction

Pink continues the Apple tradition of innovative, attractive, easy and fun-to-use human interfaces. The Pink Human Interface also has the following goals:

- Similar in style and appearance to the current Macintosh interface but more refined and attractive.
- Integrates and simplifies existing Macintosh interface features where appropriate.
- Avails users of new technologies in Pink.
- Designed to support groups of users, yet retains its focus on the individual.
- Run on existing hardware, but designed for the attachment of additional interface peripherals.

This document consists of three parts. The first section is a description of the Pink human interface design principles. This is followed by a proposed user interface for some parts of Pink. The last section describes the architecture of the interface.

Extending the Macintosh Design Principles

The user interface design principles enumerated and detailed in the book "Human Interface Guidelines: The Apple Desktop Interface" will be applied to the Pink user interface. Experience and changing technology suggest that some modifications are needed. Pink also adds two additional guidelines: "It works the way you do" and "Fun to Use."

Each of the guidelines is reviewed below. The boldface sentence at the start of each section summarizes the book's definition.

Metaphors from the real world

Use concrete metaphors and make them plain, so that users have a set of expectations to apply to computer environments.

Lisa and Macintosh used the Desktop metaphor; the underlying metaphor for the Pink User Interface is the physical world. This very basic metaphor is one in which all people are expert and so it is perfect to the extent that the illusion can be supported and applied in Pink. By generalizing to the physical world metaphor it is expected that Pink will be able to incorporate future enhancements with less degradation of the interface than we have experienced in Blue.

Pink will layer the Lisa/Macintosh Desktop metaphor (actually a document oriented, 'marks-on-paper' world) on top of the physical world metaphor. Other metaphors, e.g. production studios, workshops, and even alternate realities can also be layered on the physical world metaphor and can exist side-by-side simultaneously without basic conflict because they too have their basis in the physical world.

The use of a metaphor does not imply that Pink must be subject to its limitations. The metaphor should be extended when clearly appropriate. However, the extensions should be cast in ways that seem familiar and plausible.

It works the way you do

A new guideline but an old tag line from Lisa(?) advertising intended to convey that the computer supported users in their work rather than forcing them to work in a limited, machine-dictated manner. How do people work? They often change direction in mid-stream. They feel their way through things rather than planning everything in advance. They experiment and tryout ideas. They have good/poor memories. They are sloppy/neat. They organize things one way sometimes, a different way other times. People sometimes work alone, sometimes in groups. Some things are private, others are (sometimes) shared. Some things are worked on alone, some by several people.

Supporting flexible working styles is not the same as giving users a zillion options. While large numbers of options theoretically support extensive personalization, they also increase complexity.

The Pink Human interface extends Blue to include user-centered mechanisms to support projects, sharing, network access, and communication.

Direct Manipulation

Users should be able to manually interact with the metaphor rather than having to 'talk' to it via the keyboard.

The most important factor affecting the quality of direct manipulation in Pink is the physical similarity of manipulations in Pink to their real world counterparts. It is best to maintain a direct correspondence between the real world and Pink's virtual world. For example, moving something in Pink should always be done by the user moving the mouse, rolling a track ball, or other comparable physical displacement movement.

The limited capabilities of the hardware may make the implementation of a physically corresponding action impossible and require the creation of controls in order to support a needed operation. These controls should, when possible, simulate real world controls that are used in similar situations.

Pink is designed to make it easy to add and/or substitute I/O peripherals, e.g. stylus, touch pad, and track ball, that are more appropriate to specific tasks.

See-and-Point (instead of remember-and-type)

Users select actions from alternatives presented on the screen; they shouldn't have to remember anything the computer already knows.

Everything the user can do with the machine should be visible in some way because it is easier to recognize what's available than to recall it. Blue implements this via the direct manipulation of menus, buttons, and dialogs. Pink will address issues raised by complex application menus, large monitors, and multiple monitors which make direct manipulation of the objects slow and tedious.

Pink allows the user to capture and save direct manipulation sequences in an easily readable, editable, and reusable textual script. Scripts may also be typed-in directly by the user and may include items that cannot be (easily) included via direct manipulation. The resulting scripts can be attached to objects in the system and invoked manually or by other scripts.

Many applications in Blue have added invisible commands to the interface, usually in the form of modified mouse-clicks, using the COMMAND, OPTION, and SHIFT keys. Pink defines a mechanism to make visible these few invisible meanings.

Users can use on-line help to ask about the things they see and the way to do multi-step tasks. However, on-line help is not a substitute for good interface design.

Feedback and Dialog

Keep the user informed by providing immediate feedback.

Feedback and dialogs are the way the computer shows the user what is going on inside itself and across the network. Feedback must be immediate for direct manipulation interactions to work well. Long tasks, which can potentially run concurrently, will use animated feedback to indicate actual progress and anticipated completion when possible rather than a "time is passing" wristwatch. (How long do you look at the watch before deciding that something is wrong?)

Pink's multi-tasking will allow users to do other things while long tasks proceed. Feedback and abort mechanisms for these ongoing tasks must be accessible but not disruptive.

User Control

The user, not the computer, initiates and controls all actions.

Under Blue users must sit and wait for a long operation, e.g. initializing a disk, to finish. The only thing they can do is to cancel the operation by typing CMD-PERIOD (if supported). Users are also arbitrarily interrupted by network related tasks (mail applications are notorious for this). In Pink, users will be able to immediately select and use objects that are not part of the long operation.

The user, not the computer, controls the positioning of the pointer, icons, windows, and scrollable content. When it is necessary for the computer to scroll to bring the selection into view during an operation, it is best done in a way that helps users to quickly reorient themselves by minimizing change and/or maximizing the surrounding context.

Pink extends the concept of user control to networks by maintaining a user-centric view rather than the bureaucratic, system-centric view that so often results. While system security requirements may require the delegation of a few functions to network administrators, in general ownership and control of objects on the network are delegated to individuals.

Safety (a.k.a. Forgiveness)

Users make mistakes; make it easy for them to reverse their actions.

Forgiveness provides the user with the ability to back out of mistakes. Pink extends the single level undo in Blue to a multi-level undo and defines mechanisms that make clear any limitations.

Consistency

The skills a user learns should be transferable across the system.

Pink's toolbox-based interface elements are supported at a higher, more comlete level than found in Blue so less effort and understanding is required of third-party developers in their use. The result is that standard interface elements will be more uniformly implemented across applications.

Third-party developer needs and desires for extended functionality and Apple's inability to respond in a timely manner have sometimes resulted in a variety of application-specific ways to do something. The Pink interface will define standard mechanisms for these relatively new extensions as appropriate.

We cannot anticipate nor support every developer's needs and desires in the future. However, Pink will provide a toolbox of basic interface elements that can be combined to support future functionality.

Many applications in Blue have added invisible commands to the interface, usually in the form of modified mouse-clicks, using the COMMAND, OPTION, and SHIFT keys. Pink assigns standard meanings to these keys, e.g. 'constrain' or 'do'.

Perceived Stability

Users feel more comfortable in an environment that only responds to their actions rather than changing randomly.

Whereas Blue initially supported a single thread of execution, the user's process, and was more recently extended to include background tasks such as printing and mail, Pink must provide users with a sense of stability in a heavily multi-tasked environment.

WYSIWYG

There should be no significant difference between what the user sees on the screen and what eventually gets printed.

Pink's resolution-independent, anti-aliased graphics (Albert), and better display hardware will enable the appearance of a document on the screen to correspond even more closely with the printed appearance than in Blue.

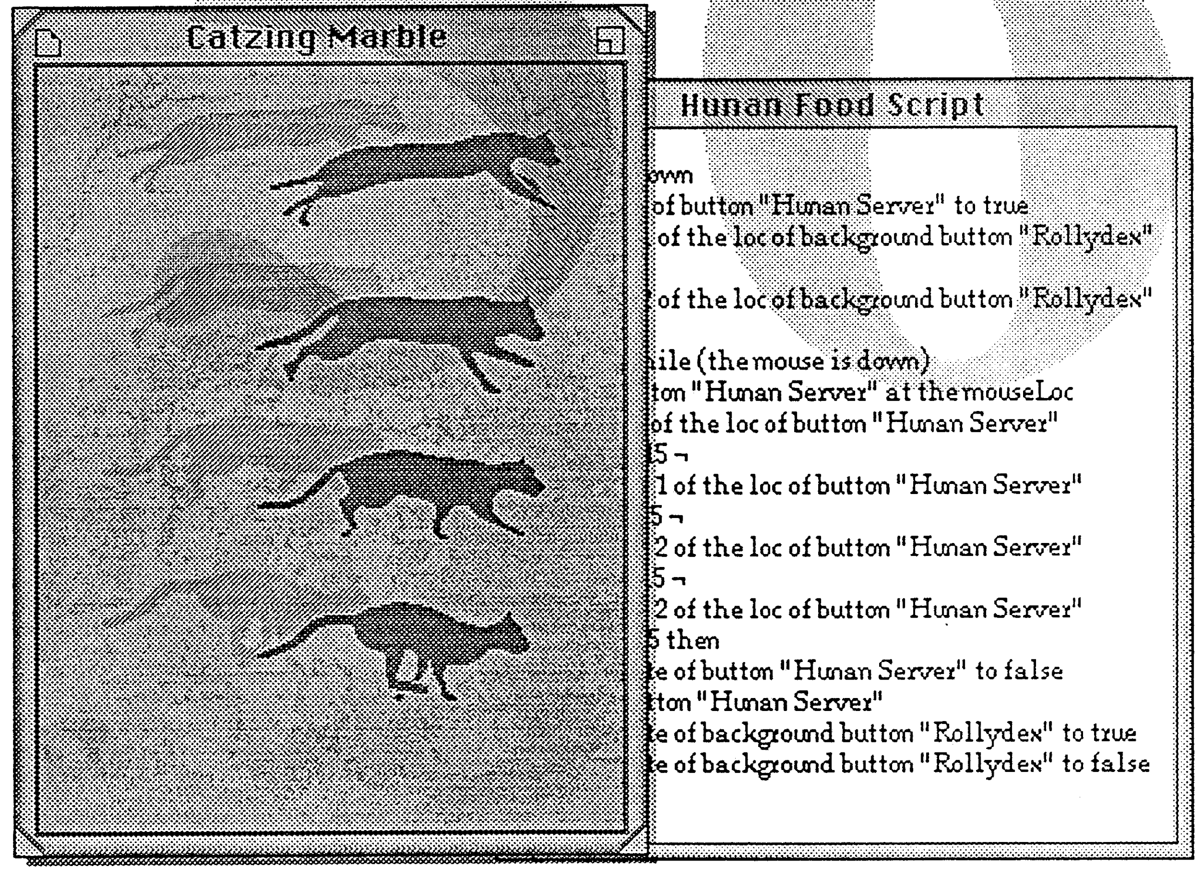
Pink also extends WYSIWYG to include sound, video, and animations that can both be manipulated on the machine and output in the form of a recording to video and audio tape.

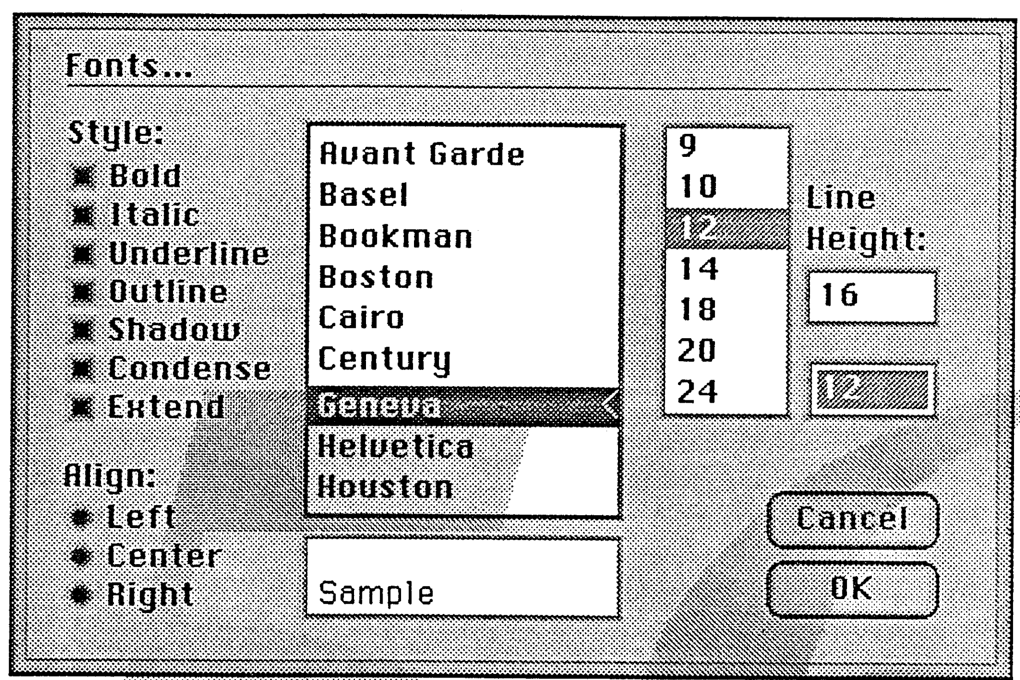
Aesthetic Integrity

Visually inconsistent displays are confusing.

Objects in Pink will have a more refined and realistic appearance than in Blue. Four-bit, sixteen color provides the design baseline. Future hardware having appropriate performance may assume 24-bit color.

Work is proceeding on a 2 1/2 D, front-light model. These two pictures of Pink windows and common interface objects are snapshots of the current state of development and should not be considered the finished product. (They also suffer in the gray-scale conversion and printing.) Certain aspects, e.g. the transparent drop shadow around the active window, may not be implementable for performance reasons.





Fun to Use

The Lisa and Macintosh User Interfaces were fun and intrinsically motivating. People who watched a demonstration were virtually compelled to sit down and try it themselves. All of the factors below, many of which are embodied in the other guidelines, contribute to an enjoyable experience.

- visible — what can be/was done is obvious
- efficient — little wasted motion, extraneous actions
- very responsive — adjustments are just that, not a series of commands
- obvious progress — feedback shows what is happening
- safety — it is easy to recover from mistakes
- sense of control — modeless, machine responds rather than prompts
- novel stimulation — pleasant, entertaining, unexpected feedback, e.g. animation
- improved results — the computer-based end product is demonstrably better

On the negative side, anything that requires extra time and effort will be irritating. This means that many things that make games interesting, e.g. non-obvious strategies and high failure rates, are inappropriate in Pink.

The only way that developers can tell if their products are enjoyable to use is to watch all kinds of people use them. User test, user test, user test!

A Pink Interface Proposal

This document attempts to describe how the Pink system will look to the user. The purpose of the design is to give us something to criticize. It should function as a "strawman" – an early plan set up with the intention that it should be knocked down.

Describing a user interface is tricky business. My approach will be to begin with the existing Macintosh interface and pare it down to a simple core. The next step is to add (hopefully) clear and simple interface parts to represent the functionality that was stripped away as well as the new functionality that defines Pink. In other words, the design requirements are:

1) Clean up the old stuff
2) Provide an interface for the new stuff (e.g. collaboration or multitasking)

A few comments might help explain the philosophy behind the design presented here. First, I am extremely wary of trying to do this at all. Any complex interface is ultimately defined only by the implementation. It's just plain impossible to predict all the cases that need to be dealt with given the flexibility of a Macintosh-style direct manipulation interface. At the same time, some large scale design is necessary to make the interface consistent and also to figure out what the user comprehendible chunks of the system are.

Second, this design is very much a minimalist interface. I have aimed for the simplest interface given the two design requirements stated above. In criticizing the various parts, I hope the question we begin with is "What can't we do with this that we really want to be able to do?"

Third, this design does not mesh perfectly with other Pink user interface designs described in Big Pink 3 or elsewhere. That's ok -- the more concrete ideas we have, the more intelligent the debate.

Finally, what you are reading represents a snapshot of work in progress. The glaring holes are marked with "***", take the menu command names with a grain of salt, and don't even consider the artwork.

With all that in mind, please read on.

Macintosh Abridged

Picture if you will MultiFinder and System version 6.0.4. Get rid of everything in the Apple menu. Pretend for just a moment as if the Chooser, the Control Panel, and DAs were all nonexistent. The plan is to add the functionality contained in these elements today in a clear and simple way given a clear and simple Finder to start from. Similarly, get rid of standard-file. (Bear with me on this...) Get rid of everything in the system folder except two files -- "System" and "Finder". (And just to be complete, get rid of the Finder menu items "Get Privileges" and "Set Startup".)

Add to this minimalist system two features from NuFinder: Keyboard Navigation and the "Find..." menu command. Keyboard Navigation allows the user to type the first few letters of a filename to jump to and select that icon in the frontmost Finder window in a manner similar to the scrolling list in the Standard File Open Dialog (the arrow keys also work). The "Find... / Find Next" menu commands very quickly search all mounted volumes for a filename, opening any necessary windows and selecting matching files. These features are needed because the Finder is going to be used to represent lots of new kinds of information, and the better it is at finding, the easier that will be.

The New Layering Model

Menubar

The menubar stays at the top of the screen. It might be a user preference to place redundant menubars at the top of each topmost monitor on a multiple monitor system.

Document Layers

The first change is to the window layering scheme. Instead of MultiFinder's one layer per application, the rule is one layer per document. In addition, each Finder window acts as if it were in it's own layer. So a possible window ordering from front to back might be:

Excel Document "My Budget"
Finder Folder Window "Release Notes"
MacWrite Document "My Novel"
Finder Folder Window "System Folder"
MacWrite Document "Letter to Mom"

Clicking on any window brings that window (and only that window) to the front. So if the user clicks on the "System Folder" window, it comes to the front; immediately behind it is "My Budget", then "Release Notes", and so on.

A document window may have attached satellite windows that appear when it becomes frontmost. Satellite windows work like the windoids and utility windows of today. Windoids always float above the document they are attached to (like a tool palette in a graphics program), while utility windows are ordered within a document layer (like the "Find/Replace" window in a word processor).

The Windows Menu

Under a menu labeled "Windows" (or perhaps dropping down from the MultiFinder icon on the right side of the menubar) is a list comprised of the word "Finder" followed by the names of all open documents (arranged alphabetically). If the frontmost window is a document, the last menu item is a "Hide" command. The "Hide" command works just like "Set Aside" MultiFinder except that it hides the current document instead of the current application. So, for the example above, the Windows Menu looks like this:

Finder

Letter to Mom
My Budget
My Novel

Hide "Letter to Mom"

Like the current Apple menu, small icons appear next to each item in the list of documents to convey more information and a check mark indicates the frontmost document.

The Apple Menu

The Apple Menu contains the "About" menu command followed by a list of tools and documents to which the user wants quick access. The mechanism for customizing the Apple Menu is described later in the document. For the moment, the Apple Menu might look like this:

About Excel...

Alarm Clock
Calculator
Puzzle

Random Thoughts
Doodles

Again, small icons appear next to each item to indicate type. Note that the documents "Random Thoughts" and "Doodles" are not necessarily currently open. When a document in the Apple menu is open, it's name remains in the Apple menu as well showing up in the Windows menu.

Fast Switch

Choosing "Finder" from the Windows menu brings all the Finder windows to the front. A command key equivalent would be nice, since this will be a fairly common operation. Even nicer would be some background preflighting of this operation so that the user could bring the Finder windows to the front almost instantaneously. In other words, if there is processor time and memory available, prepare the bitmaps for the Finder windows in advance so that bringing the Finder to the front would happen very quickly, without having to wait for all the usual update event processing. The hope is that the ease and speed of this operation would obviate using Standard File to open documents.

Icons on the Desktop

Placing icons on the desktop in the current system is troublesome for two reasons. First, they get buried under other windows. Second, given a file on the desktop, it is difficult to know or predict on which disk that file is actually stored. We can solve the first problem to some degree by establishing stricter rules for opening document windows so that the right edge of the desktop is always left uncovered. We may invent some kind of icon dock which floats above documents, but for the moment leave icons on the desktop as they are (pre-7.0).

The New Document/Application Scheme

Stationery

New documents are created using the long familiar stationery pad. Double clicking on a stationery pad results in a quick animation to show an untitled document sliding off the stationery pad. This icon immediately opens into window named "untitled". The user can name the document by choosing "Rename" from the File Menu. (An option in the naming dialog makes this document into a stationery pad itself.)

The user can also name the document by going back to the Finder while the document is open, selecting the icon, and typing a new name. Just as the titlebar in a Finder window changes dynamically to reflect a new name as it is typed, the titlebar in a document window changes dynamically to reflect the new document name as it is typed. The rule is that an open document window is always in sync with its finder icon. If the user moves the icon of an open document into a different folder, the document is simply now in the new folder.

There continues to be a "new" command in the File menu of applications. It creates a new untitled document from the stationery pad that was used to create the current document. Double clicking on an application icon will open a window containing, among other things, a default stationery pad that cannot be deleted.

Versions

Document saving is handled quite differently than it is in the current system. Documents are always saved to disk to within an operation or two. It's as if the user were constantly hitting command-S. Multiple level undo and versions will (hopefully) replace the back tracking allowed by "Revert to Saved" in the current model.

Undo and Redo step back and forth through the operations in a single document. What exactly is defined as an operation is left up to the application, but Apple should offer suggestions at some point about questions like "does selection count as an operation?" It seems as if the only way to answer these questions is through experimentation.

At any time while editing a document the user can choose to mark the current state as a version ("Checkpoint," from the File menu). Also at any time while editing the user can choose to revert to the previous version ("Previous Checkpoint" from the File menu). This operation is, of course, undoable. (It may be a user option to create a new version each time a document is opened.)

The Get Info Box in the Finder has a list of the past versions of a document with the date and time of each version. A version is not a separate and independent document but rather a snapshot. The user can select a past version from this list. Available operations include

1) name the version
2) delete the version
3) create a real document from that version, i.e. make a new, editable document which then has its own ongoing version history.

Note: Although at first glance it might seem appealing to allow the user to just drag a version out of the Get Info Box and onto the desktop, I think it is important to maintain a clear distinction between Finder windows (which contain only named icons and whose only semantic is containment) and other kinds of windows (which contain other information and always have additional semantics). There are many things about the Get Info Box that are different from a Finder window, and so it is essential to make the "create a real document from old version" operation explicit (i.e. have a button "Resurrect Version" in the Get Info Box) rather than leaving it up to the unprompted dragging operation.

(Version history may deserve it's own menu command "Get Version History" and it's own "Version Info Box" instead of overloading the already full Get Info Box.)

(***Does the user get the undo history for old versions?)

Summary - The File Menu

Suppose the user had just opened a document from a stationery pad labeled "Pink Letterhead." The File Menu would look like this:

New "Pink Letterhead"
Rename "Untitled"
Close "Untitled"
Checkpoint
Previous Checkpoint
Print

Tools - Applications Without Documents

There are some icons the user will open which do not yield documents. Call these things tools. Into this class will fall the desk accessories and control panels of today. The rule is: there are no mysterious chunks of code that don't open into a window.

Multitasking

To understand what multitasking will do to the system, join me in the following thought experiment. Picture a simple animation application. There are two squares in a window. The user selects one and chooses "Spin". The square starts spinning. Then the user selects the other square and chooses "Spin". Now both squares are spinning -- voilà -- multitasking. This is the kind of stuff my mother could understand. What is it about this scenario that makes it so easy to grasp? There are multiple tasks, or activities, going on simultaneously but each one is confined to a single object. The rule, then, for multitasking is this:

ONE OBJECT, ONE TASK.

What is an object? In general, an object is a document or a tool. This means that the user could see, say, three documents, each one doing something. But, like the spinning squares, each document is only doing one thing at a time. (Depending on the application, there may also be multitasking objects within a document. So the animation application above is introducing a finer granularity of multitasking objects with the spinning squares. However, what goes on inside documents is not the focus of this discussion, so assume that the objects are documents tools.)

What does all this mean? Picture MultiFinder. With multitasking, the user will always always always be able to click on another window and immediately switch to that window. The user will always always always be able to pull down the menus, drag a window, or close a window. Modal dialogs in the current system will be modified to be movable, and non-modal if possible. Dialogs and alerts will be associated with documents, not applications. They should appear on top of and visually connected with the document to which they refer. As such, they can be temporarily ignored by switching to a different layer. Finally, there will be some visual indication for documents that are busy and this will appear in the icon, in the titlebar of open documents, and in the small icon next to document names appearing in the Windows menu.

Mouse-ahead

A somewhat separate issue is whether to allow commands on a document to be queued up. For example, consider a spreadsheet that is busy recalculating. Three things could be done with the "Close" command in the File menu (remember, menus are always accessible).

1) Disable it. The "Close" command (and almost every command) is grayed out.
2) Leave it enabled without command queueing. When chosen, the "Close" command would cancel the recalculation (reverting the document its state before the recalc) and immediately close the window.
3) Leave it enabled with command queueing. When chosen, the "Close" command would close the window after the recalculation is done.

In the spirit of making decisions, assume that the rule is this: whenever possible leave a menu command enabled with queueing (3), and when that can't be done, disable the command (1).

Status Window

The "System Status" tool opens to show a list of currently busy documents and tools and what each one is doing. A task would be listed in "System Status" only if it is going to take longer than a second or two. If possible, some indication of estimated time remaining would be nice.

***

Cancelling a Background Task

Any task can be cancelled from the "System Status" tool. Also, the task in the frontmost document or tool window can be cancelled using command-period. Lastly, the user can select the icon for a busy document or tool and cancel the task, again using command-period. What happens to queued commands when the current task is cancelled? They get cancelled also. Cancelling always returns an object to its non-busy state.

Transparent Multitasking

Finally, some parts of the system may use multitasking but never tell the user. For example, the Finder windows preflighting described above would probably be done using a separate task, but the user will never know about it. The rule is: behind the scenes uses of multitasking are never noticeable to the user. (Of course -- if they were noticeable, they wouldn't be behind the scenes, would they?)

Examples

Copying files - ***

Printing - ***

Formatting disks - ***

Notification

***

The Network

On the right side of the desktop, beneath the icons for the user's hard disk drives, there is an icon labeled "Network." Opening this yields a window containing folders labeled "People," "Printers," and "File Servers". These folders and their contents are called, for now, "network things." The Network behaves similarly to read-only disk drive. Anything the user drags out of the Network is copied, not moved to the new location. The user can rearrange network things within a network window and change the view but can't permanently change the Network. The reason the Network can't be modified is that all network things represent something in the real, physical world. The way windows open from the Network icon is determined by the way people and equipment are connected in the real world. The user can't drag a printer from inside the Network icon to the trash and forever lose that printer. The user can't move a fileserver from one zone to another. Network things would get a different "look" than ordinary file system objects to convey they fact that they cannot be edited.

Note: The following scheme is based on the assumptions that zones are a fact of life.

Copying Network Things

Since the Network works like a read only disk drive, dragging a network thing out of a network window makes a copy of that thing. So I can drag a person, a printer, or a file server to the desktop or to any of my own folders. The copies of network things work just like network things found inside the Network icon.

People

Within the "People" folder, organized by zones, are icons for other users on the network. Double clicking a person gives useful information about that person such as their phone number or office location. People icons don't do much by themselves, but are used in other parts of the interface (such as mail) to refer to other users. See Figure 1.

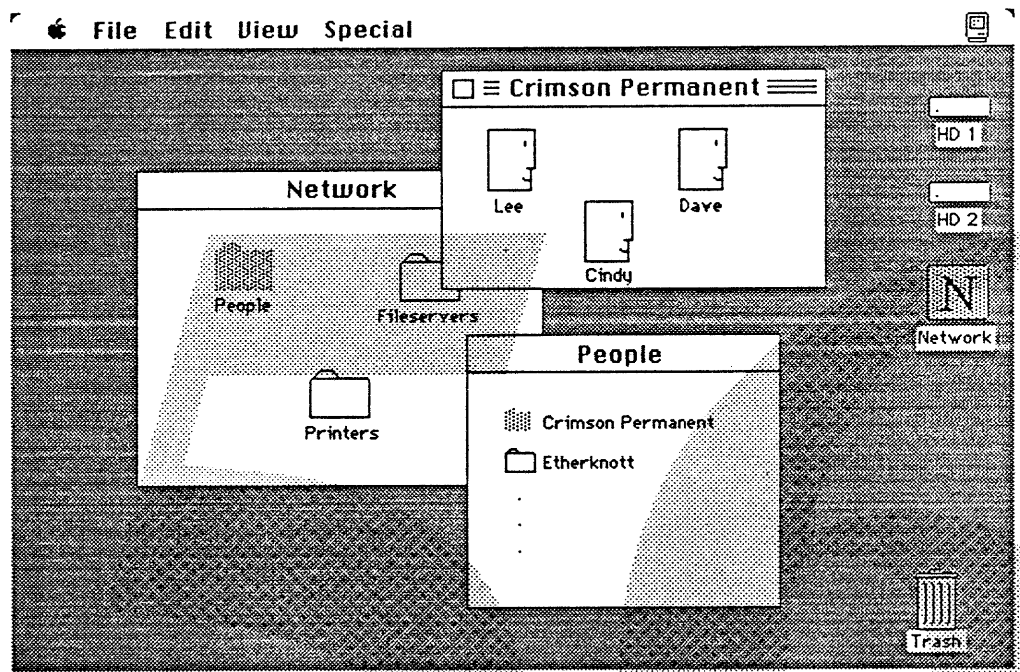


Figure 1

Printers

Printers are also arranged by zones. In a given zone, the view menu allows the user to see printers as icons or sorted by name or kind. With some setup by a system administrator, a "view by location" might present printers as part of a map of showing their geographical location.

The user can print a document by dragging its icon to a printer. The document icon goes through some quick animation with the printer icon, then the document goes back to where it was dragged from and the print dialog appears. The print dialog, like most good Pink dialogs, should be non-modal. With the fast switch to the Finder described earlier, printing by dragging documents to printers should be fairly painless, but some mechanism for printing from within a document is still necessary. This requires defining a default printer. For more details on how that is done, see "Print Shop" below.

Double clicking on a printer opens a window showing printer status.

File Servers

Once again arranged by zones, icons for file servers appear inside the Network icon. Call these icons "server stubs." Double-clicking on a server stub mounts that server. Server stubs work very much like Quick Mount documents on the Mac today. Although it would be nice to just open the fileserver's window directly from the stub icon (instead of mounting the server), this leads to several gotcha's that are best avoided at this point. No doubt there is a better solution than the one proposed here (a

mysterious chunk of code which doesn't open into a window). Like people and printers, the user can drag a server stub out of the network icon and leave it on my desktop or put it inside a finder window.

Double-clicking on a server stub brings up a password dialog box for file servers with controlled access.

Other Kinds of Network Things

Other devices on the network also appear inside the Network icon.

***How do they work?

Modems

Scanners

Direct Manipulation Cut, Copy, and Paste

The traditional menu commands for cut, copy, and paste continue to function as they do today. In addition, these editing functions can be performed using direct manipulation. Most users are familiar with dragging the selection around in Paint and Draw applications. The Janette prototype illustrated that selected text can also be treated as a draggable object. Hopefully, "draggability" can be added to spreadsheet cells, sound and animation clips, and most kinds of data found in documents. The general rule is: dragging moves an object and option-dragging leaves a copy behind.

If, while dragging something around, the user hits the edge of the window, the document auto-scrolls.

In order to allow dragging from one document window to another, magic "portholes" are added to the window. (See Figure 2.) If, while dragging an object in one window, the user moves the pointer through that window's porthole, the object reappears outside the window in the form of a small clipboard icon. The user can continue dragging this icon about (it floats over everything else), and then enter the porthole of any other document which understands the clipboard's data. As soon as the user drags the clipboard icon into the porthole of second document, that window becomes the active window and the object can be placed as usual. If the user stops dragging the clipboard icon around before entering another document, the clipboard icon remains, floating in a layer on top of everything else in the system. In fact, the user can leave as many of these clipboard icons around as desired. At any time they can be dragged (or option-dragged) into the portholes of open documents.

If we discover there is no simple way to incorporate portholes into the window frame, it may be appropriate to make the portholes appear only when the user is dragging an object.***

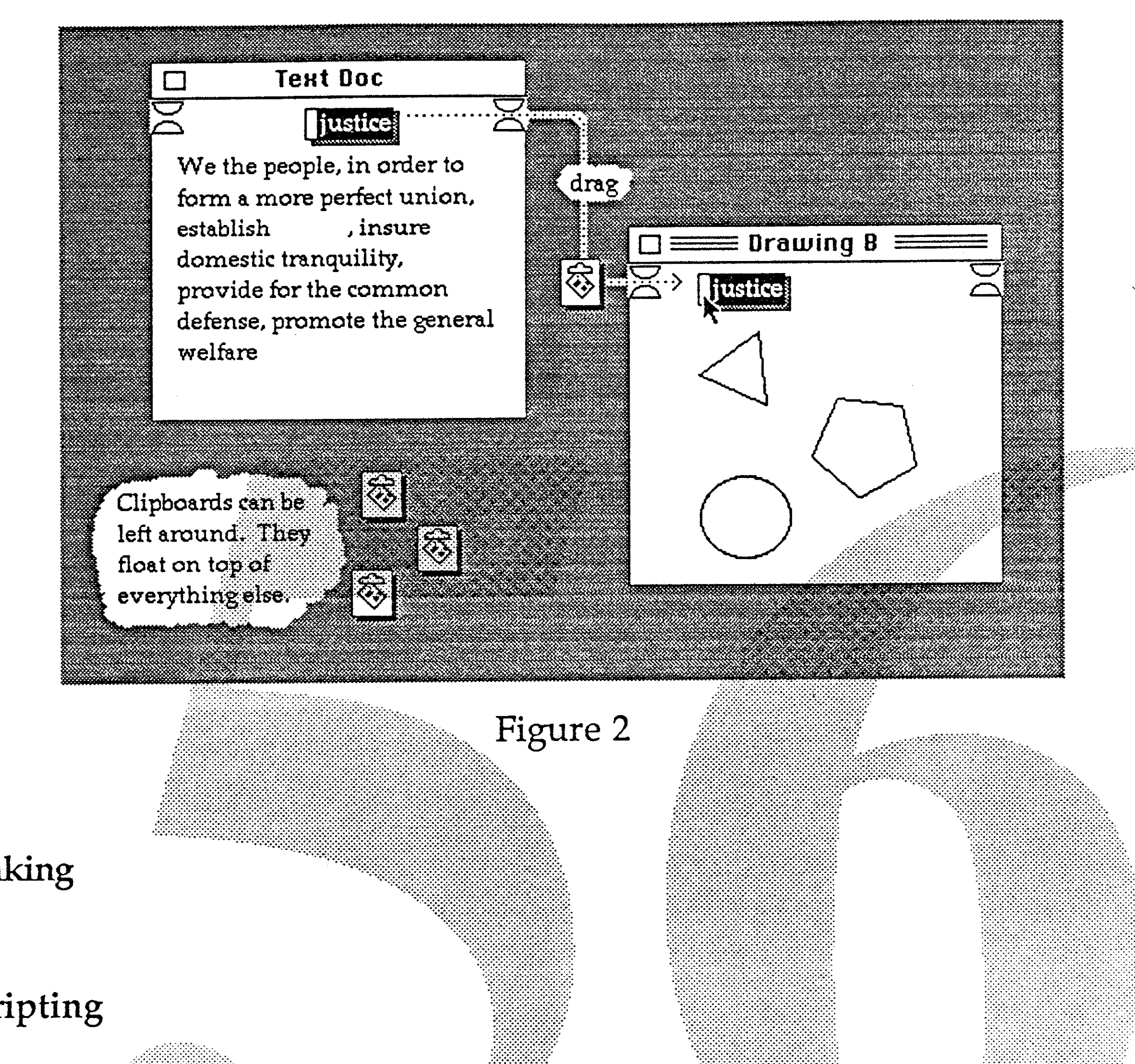


Figure 2

Linking

***

Scripting

***

History Window

*** Connection to System Status Window?

Post-its

Post-its or annotations are kept very very simple. An "Attach Note" command is added to the standard edit menu. The user can attach an annotation to any selection. A small standard icon appears in the document. Double clicking on it brings up a window. There are buttons to record and playback sound, and a simple type-in text field. The text field would be limited in powers to approximately that of TeachText.

Filling Out Forms

The designs for the next few pieces of the interface apply a new technique to expand the use of direct manipulation. In the current Finder, direct manipulation (or dragging) of icons representing files has two characteristics which make it successful: First, it always works. The user doesn't have to know any special rules about where dropping an icon is allowed and where it isn't. Second, placement of icons

by direct manipulation always means the same thing -- where a file is stored. In fact, the place where this meaning is overloaded is exactly the place where users get confused. The System Folder in the current model has all kinds of additional semantics and as a result it has become somewhat of a sinkhole in the Macintosh interface.

It's time for a new widget. If dragging icons from the Finder is going to be used to represent some of the new functionality of Pink, users need to know when that dragging is appropriate, which icons go where, and what the dragging represents. The mechanism for all this is called (for lack of a better name) "filling out forms". The idea is that in order to complete a task which involves something represented by an icon in the Finder, the user drags the icon into the appropriate blank space on a form. The outline shape of a blank space in a form matches the outline shape of the kind of icon that goes in that space. Most forms have the magical property that there is always a blank space available. When an icon is dragged into an existing blank space, that space is filled and the eternal blank space moves over one.

In addition to indicating when dragging is appropriate, forms are the mechanism for referencing things represented by Finder icons. When an icon is dragged into a form, the thing it represents is not moved to the form nor is it copied to the form. Instead, the filled in blank is a reference to the thing. And, filling in blanks is the only way to create references to things. There is no general "alias" command as in System 7.0.

Fill-in-the-blank forms are used to solve a whole series of interface problems. The following example works through the interface for sending mail in some detail. Most of the interfaces that follow after this section also use this technique.

An Example - Sending Mail

The easiest way to explain the "filling out forms" idea is by example. Consider the problem of sending e-mail across the network. In order to send a mail message to another Pink user, the user needs to refer to at least two kinds of Finder icons:

1) the recipients of the mail message (recall that people can be found inside the Network icon)
2) any files to be enclosed with the mail message.

The user double-clicks on the mail stationery pad and a window like Figure 3 appears. Then the user successively drags people to the recipient blank (shaped like the outline of a person icon) and enclosures to the enclosure blank (shaped like the outline of a generic icon, since anything represented in the Finder can be mailed). As an icon is dragged over a blank on a form, if the icon can be used to fill in that blank, the blank highlights. Once an icon has been dropped into the form, the original icon remains in place and a copy appears in the blank. This form is not all fill-in-the-blank though; there are simple type-in text fields at the bottom. Figure 4 shows what the form would look like after two users and an enclosure had been plugged in. Clicking "Send" would send the mail. Closing the mail document would allow the user to save this document or discard it, just like any document.

Notice that dragging a person or file into the mail form is not the same as dragging an icon into a folder. Dragging an icon into a blank on a form is a reference to the real thing in the file system or the network. So the icon for the enclosed file "Budget Doc" does not represent a copy of that file, but a reference to that file.

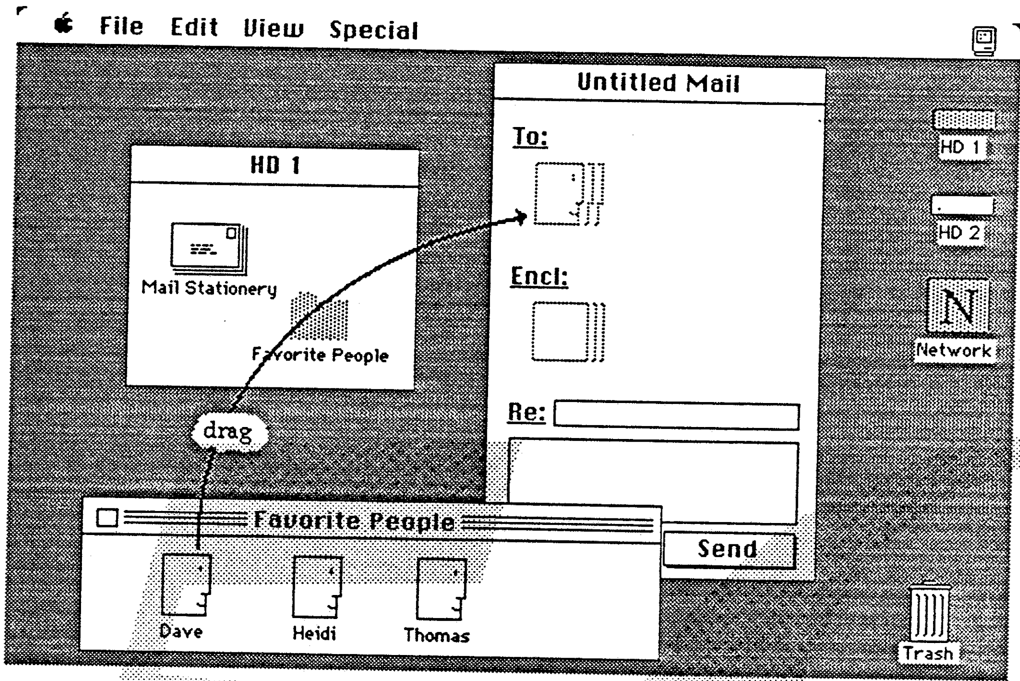


Figure 3

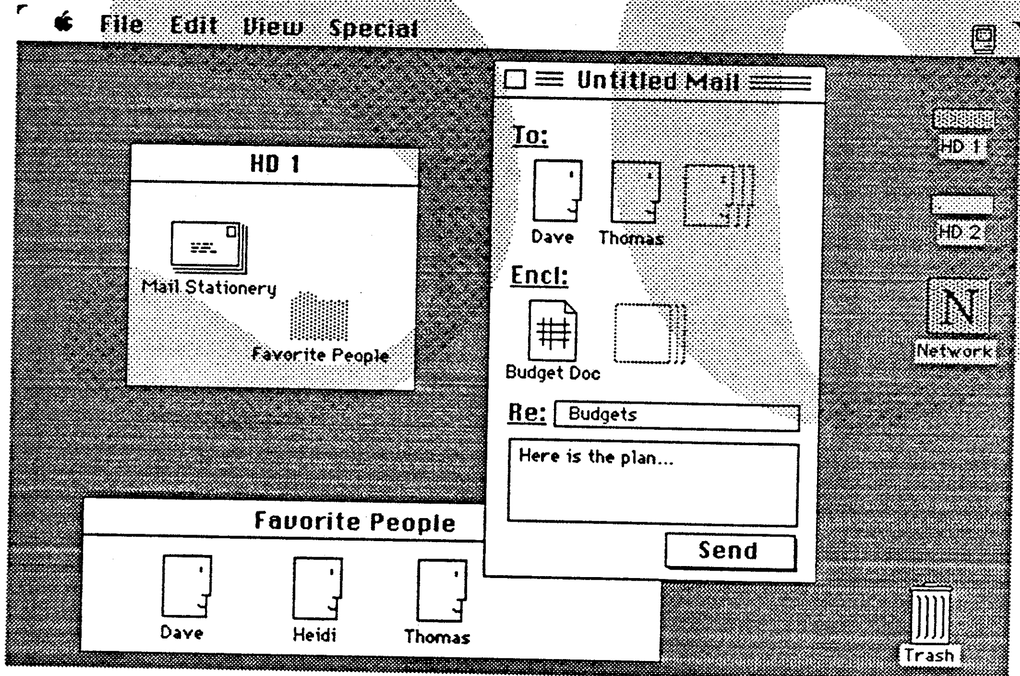


Figure 4

Summary

The blank on a form tells the user three things, and these hold true for all blanks on all forms:

1) that an icon of a particularly kind can be dropped here.
2) that dropping an icon here has semantics other than the standard Finder semantic of rearranging my file system. The semantics are stated right there on the form, probably as text. This should visually distinguish a window containing a form from a standard blank background finder window.
3) that the icon being dropped here is not being moved or copied but referenced.

Details

How does the user remove something from a form?
Drag it to the trash, or perhaps just drag it outside the window.

Can the user drag an icon out of a partially completed form and onto the desktop?
No. Since being in a form implies that the icon is a reference, and since forms are the only place where references are allowed, the user cannot drag things out of a form onto the desktop or into Finder windows.

Can the user drag an icon out of one form and into another?
Yes.

Can the user rename an icon once it's inside a form?
No.

Can the user get from the icon in the form to the real icon, i.e. navigate from the reference to the real thing?
Say no, for now.

What happens when there isn't room on the form for all the icons that have been filled in?

Solve this problem in the same way as Finder windows solve it. The user can grow the window, scroll, or (perhaps) different views of forms are allowed which parallel the different icon views in Finder windows.

Where exactly can forms appear?
Inside document windows and inside tool windows.

56

Groups of People

Opening the Group stationery pad gives a form similar to Figure 5. The user successively drags in people icons to create a group. Click on the close box, select the untitled group document, and name it. Group icons resemble people icons, and can be dragged into any blank that is expecting people (including the group form itself – groups of groups). Groups are first order objects just like people, so the user can duplicate them, mail them around, etc.

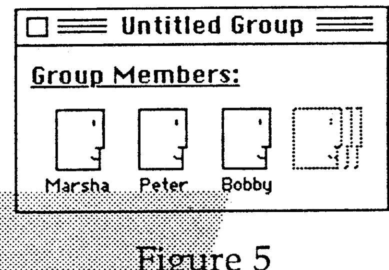


Figure 5

Collaboration

Three kinds of collaboration have been discussed:

1) Projector - style file access management
2) whole screen sharing
3) document sharing

There is no design yet for the Projector interface. Presumably the user fills out some forms to set up the document for checking in and checking out.

There is no design yet for whole screen sharing.

To begin a same-time same-document collaboration, the user would open the Collaboration tool (perhaps stationery, so collaborations could be saved). See Figure 6. Drag in the documents to be shared and the people to collaborate with and click the "Contact Everybody" button. This opens the specified documents if they are not already open and politely sends notifications to the other users. If a requested user decides to join, then the collaboration window and the document windows appear on that user's computer. The bottom part of the collaboration is a typing free for all, similar to Quick Conference in Quickmail or those chat programs on mainframes. Hopefully the collaborators will be in contact over the phone as well.

Once the collaboration has started, how should pointer passing work? This is undetermined at this point. Experimentation is the only way to find out.

Notably and intentionally missing are any control enforcing mechanisms. Everyone in a collaboration is on equal footing, and there is no access privilege scheme. If a user is afraid that collaborators will mess up a document, then that user can make a copy of the document before entering the collaboration. Also intentionally missing is any kind of whiteboard. If users want to share a drawing space, then they can open a draw or paint document.

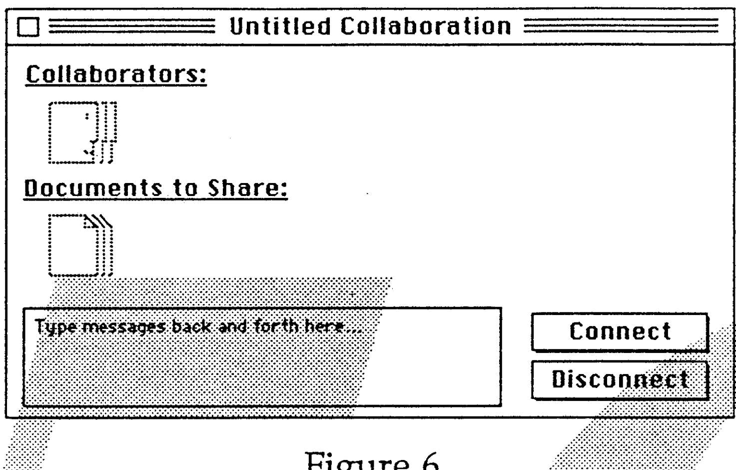


Figure 6

Project Lists

Projects are a means for collecting together a working set of documents outside the neat hierarchy of the file system. Projects could be opened, closed, or set aside all at once. The Project List will hopefully satisfy the needs that resulted in aliases in System 7.0.

Fill out a form. Connection to Projector style collaboration?

***

Print Shop

Users would drag printers into this form to set the default printer. The default printer is the printer that is used by the Print menu command from within a document. It may make sense to set a default color printer, a default black and white printer, a default legal size paper printer, etc.

*** Where to put page setup options and what is the relationship between documents, page setups, and printers?

All the Other Stuff Now Found in the System Folder

The plan is to have no magic places in the file system. Instead of moving a screen dimmer INIT to the system folder and restarting in order to get it to work, the user would double click on the screen dimmer icon to open a control panel with an on/off switch and perhaps some other controls. Switch the screen dimmer on and close the control panel. In other words, the two rules are:

1) It doesn't matter where the user puts an icon.
2) Every icon opens into something.

However, when icons can be grouped together with some higher meaning, then the device to use is the previously mentioned "Fill in the Blank".

Here are some examples:

How might the user customize the Apple menu to contain the Calculator, Alarm Clock, and nothing else?
Double click on the "Apple Menu Shop" icon and fill in the blanks. See Figure 7.

How might the user preview the font "Galliard" (a la Keycaps)?
Double click on the icon labelled "Galliard" to open a window showing the font mapped onto the keyboard.

How might the user add the font "Galliard" to the system?
Double click on the "Font Shop" icon and add "Galliard" to the system fonts form by dragging in the icon for "Galliard."

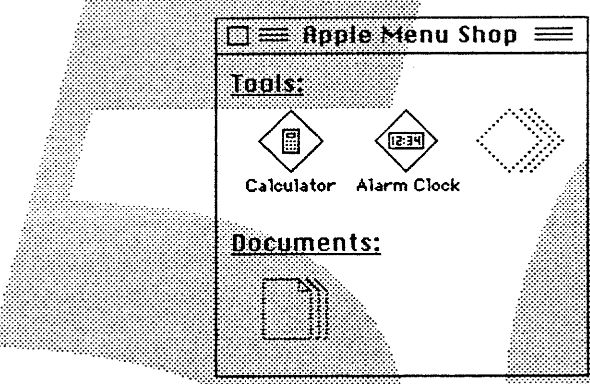


Figure 7

New Cool Things That Are Small or Less Than Pervasive

Many parts of the interface remain. Hopefully the parts that follow are more discrete than the preceding sections. That is, the user will see them only periodically and their impact on the rest of the system is clear and limited. What am I trying to say? In my mind I can see these things as relatively independent parts, separable from the tangle of concepts which make up the interface described above. I can imagine sitting down and working through the details of these problems given the framework described above without reverting to that completely fuzzy state of mind in which none of the interface is defined and everything depends on everything else. Which is also to say that these are the pieces which I haven't thought much about except to suggest that they sure would be nice. Everything in this section deserves a "***" for lack of content. With that in mind, here is a grocery list of new interface things:

Mail

Outgoing

See Figure 4 and the description above (in the section describing forms).

This represents the simplest case. Other fields, like "CC:", might be revealed using the progressive disclosure device described below under "New Widgets for Applications." Since mail forms originate from a stationery pad, there should be a mechanism for creating custom letterheads.

Incoming

Add a mailbox icon on the right edge of the desktop. The icon changes appearance when mail arrives. Double clicking on the icon opens a window containing incoming mail.

Personal AppleShare

Users would open a File Sharing tool and drag in people (or groups) and file system objects to be shared (a disk volume, a folder, or just an individual document could be shared). Those people would then see a server stub in their Network icon.
* * *

New Widgets For Applications

Here are some user interface support packages Apple provides to applications in addition to what is traditionally considered part of the toolbox.

Progressive Disclosure Dialogs

Progressive disclosure is the name for the technique of hiding complexity from beginning users. The mailbox flag in the Alarm Clock DA is the perennial example of progressive disclosure. Applications would have a standard widget to put in dialog boxes so the user could switch between the simplest version and the complex feature-packed version.

Hierarchy Viewing

In addition to upgrading TEXT and PICT to more powerful formats, it would be nice if outlining became a standard data type.

Standard Pickers

The current system has a standard color picker. In addition to improving that interface, Apple provides a standard font and style picker.

The Parts Bin

Support for organizing parts and dragging them across windows would encourage developers to use lots of direct manipulation too. See the Hoops prototype for an example of a Parts Bin.

Standard Sound Recording and Playback Interface

Standard (and Replaceable) Dictionary and Thesaurus

Provide a mechanism and interface so that the same dictionary can be used across all applications.

Dialog Layout Rules

Decide on some set of rules for the size, spacing, and arrangement of buttons in dialogs and give developers the tools that make it easy to follow those rules.

Content Based Document Retrieval and Filtering

Fill out a form using a graphical query language.

Smart Icon Cleanup

Take a better guess at what the user is trying to place in rows and columns. Make sure icon names don't overlap.

Smart Window Placement and Dragging

It seems as if there is some set of rules for usually doing the right thing when dragging windows. For example, if there is only one document from an application open, then it probably makes sense to maintain the relative positions of satellite windows when the document window is moved.

System-wide tiling or stacking of document windows might be done through a universal menu command in the Windows menu

Better Error Messages

Notice if the user is repeatedly getting the same error message and provide more information.

Shrunken Documents

Replace the inside of an document icon with a reduced version of the first page of the document.

Preferences

It is unclear as to whether user preferences should be associated with an application, a stationery pad, a document, or with an icon representing the user.

Help

Bubble help and/or procedural help.

Backup

Connection to versions?

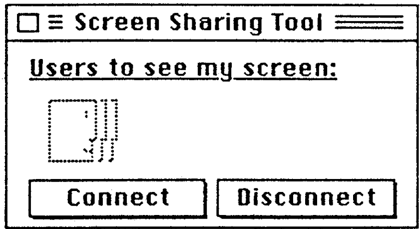
Quiz

If this was at all a coherent description of a reasonable approach, then given a problem, you, the reader, should be able to see a solution which fits the model. And your solution should be similar to mine.

So here is the problem:

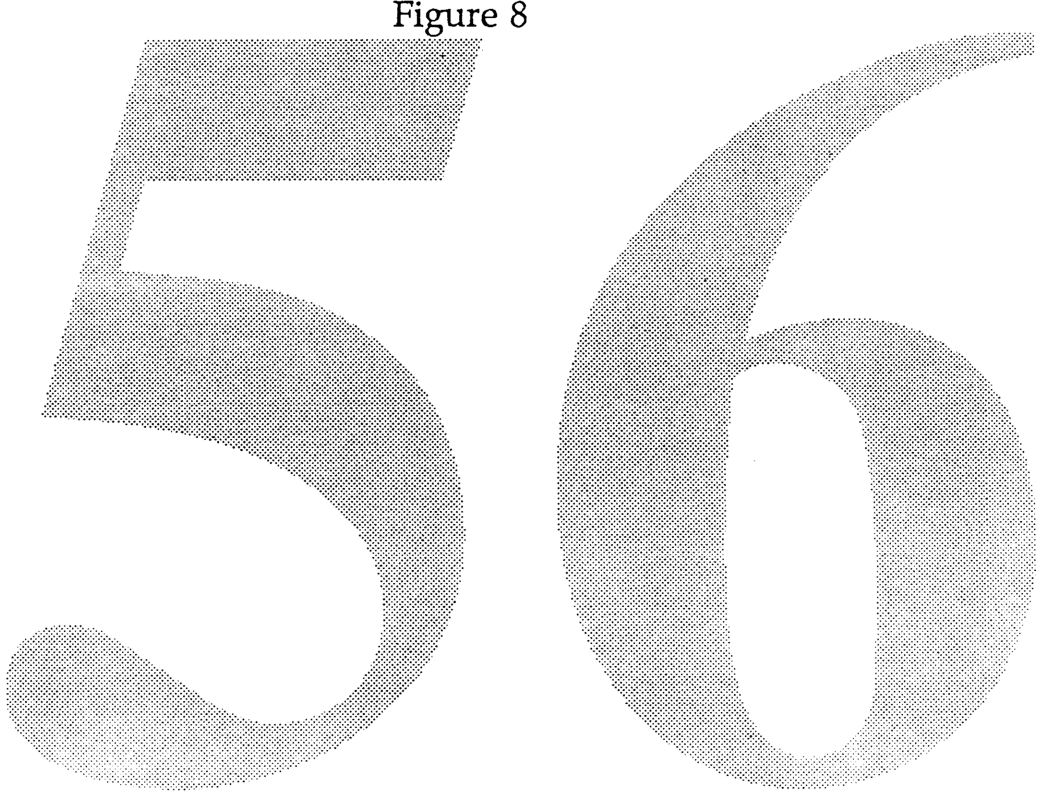
Design an interface for starting the Screen Sharing collaboration. The user needs to connect to one other user. (Don't worry about all the pointer passing conventions once the connection is made.) Design the interface for making the connection and ending the connection.

See Figure 8 for the answer I was looking for.



Double click on the "Screen Sharing Tool" to get this window. Drag in a user and click "Connect."

Figure 8



The Architecture of the User Interface

[The following has not been widely reviewed and the reader may find it incomplete and/or idiosyncratic. Reader's comments are appreciated.]

The architecture of the Pink Human Interface lays the foundation for everything involved with the user's interaction with the Macintosh. Based on a real world metaphor, enhancements to the original Lisa/Macintosh architecture, and an extended "every-user" model, the Pink Human Interface Architecture is a broader and more general architecture that supports current useful extensions to Blue, anticipated Pink extensions, and, hopefully, most future unexpected extensions.

The Pink Human Interface Architecture provides a systematic basis for designing all user interactions with the machine. Without a clear comprehensive architecture, an interface will be cluttered, inconsistent, and disorganized, inaccessible to most users.

This section contains a brief description of the basic elements of the Pink Human Interface, object, operation, and interaction, and a taxonomy of the generic objects, operations, and interactions supported in Pink. While it is not technically an architecture, that would detail all objects and their operations, what follows is a necessary first step. Also, it does not provide explicit designer's guidelines for the interface - that will have to be developed over time and involves a great deal of user testing - much of that model can be easily implied from the architecture and the foregoing strawman.

Object

Objects are the things in Pink the user works with, the "nouns" of the system. Most objects represent things in the world around us: documents, tools, people, communication mediums, ideas, etc. The rest of the objects are the made-up things in Pink that do not normally exist in the physical world, such as scrollbars.

There are four characteristics that describe each Pink object: behavior, content, connections, and interface. All objects have, in varying degrees, some or possibly all of these four characteristics. (Lisa and Macintosh/Blue defined objects that could be described by these characteristics, but by focusing on the objects rather than the characteristics, they failed to anticipate new kinds of objects, e.g. HyperCard stacks, annotations, and aliases.) The remainder of this section describe these four characteristics in more detail.

Behavior

Behavior is what things do, how they act. All objects appear to the user to have behavior. Even documents that have no code in them exhibit behavior when they are moved about (Finder code) or opened up and edited (application code).

Lisa and the early Macintosh considered only one kind of behavior, built-in. Experience and technology show that there are more aspects of behavior that need to be considered:

- built-in non-modifiable, except via preferences, comes with an object
- system-based uses system-supplied behavior when possible or overrides with built-in.
- component non-modifiable functionality that can be added to an object (may have preferences)
- script user-modifiable functionality that can be added to an object (may have preferences)
- dependent may vary by user, time, situation, editing vs. run
- cooperating the behavior of one object may trigger the behavior of another
- external behavior that is applied to other objects, e.g. move and delete; all the above are 'internal'

A task denotes the execution of some initial behavior and the behaviors it triggers. The current Pink position is that an object can only exhibit one behavior at a time, i.e. one task per object. The consequences of this position are under consideration.

Content

Content is the information an object contains, often a composition of other objects. The atomic objects are text characters, graphic shapes, pixels, sound samples, and behavioral components.

While the Lisa and the early Macintosh considered many of the aspects below, several were not.

- object relative e.g. size and creation date
- behavioral e.g. preferences, scripts, components, and help
- user task e.g. text in a document
- presentation e.g. font, face, (frame) rate, and color
- view-specific e.g. window location, size, magnification, scroll position
- app-specific only the application can work with it
- system-supported the system software can display, print, copy, and possibly edit
- kind text, graphics, bitmaps, pictures, sound samples, etc.
- organization sequence, parallel (includes time), nested, branching, spatial
- medium paper (page, sheet, cards, etc), 'tape' recordings, multi-media
- dynamic changes with time, situation
- shared multi-task access, access controls
- versions multiple snapshots are supported

Connections

Connections indicate relationships between objects. Each object is responsible for supporting connections between the objects it contains. The simplest connection is sequential: character strings, lists, cells in a spreadsheet, frames in a video sequence, sound samples in a recording. Containment is another simple form of connection e.g. documents in a folder or pictures in a document. More complex forms may require computational support e.g. cross references, picture-to-text (to keep them on the same page), spreadsheet formulas, and filters.

Objects are responsible for supporting connections from the outside to internal objects. The connection may be maintained by either unique id or attribute matching as appropriate.

A connection may be represented by a concrete object (explicit) or not (implicit). Examples of explicit connections include links, aliases, annotations, and filters. Examples of implicit connections include the document to app (document type), cross reference look up, and containment of documents in folders. Explicit connections are objects in their own right and can have their own content, behavior, and interface.

Lisa and the early Macintosh supported only a few implicit connections: containment (disks and folders contain folders and documents), sequence, spatial, and document-to-application. The early Mac also

supported name-to-object and the newest versions support aliases and warm-links. The various aspects of a connection include:

- destination — identification of the referenced object
- fan-out — can there be more than one destination
- computation — the destination may be computed at look up
- kind — an optional link type

Other items that might be associated with links, e.g. direction and behavior, are actually parts of explicit connection objects.

Interface

Interface is that part of an object that people and other objects deal with. Each Pink object has a concrete visual interface. Some objects have other representations as well, e.g. audible, printed, storage, etc. The various aspects of an object's interface include:

- external visual — simple objects are characters, lines, and pixels. Complex objects are shown as icons
- external audible — designed for visually impaired or to get user's attention
- external other — generally ignoring smell, temperature and pressure, but tactile might be used for the visually impaired.
- external electronic — for transmission and storage
- external operational — for programmatic access and control
- identification — unique id, collections of attributes, 'touching' a representation
- internal — access to the objects inside
- DM interaction — for manual user access and control
- Symbolic interaction — for symbolic user access and control

Operations

Operations are the computer commands that users invoke to get their work done. This section deals with operations and what they consist of, but not with the interaction used to invoke the command since there may be more than one interaction to do it. (However, some interactions are specified as examples.) Interaction is covered in the next section.

We can use verbal (symbolic) person-to-person communication to model this activity since that is the way we interact normally. (This is based on English, but I assume that the structure holds up in other languages given my very limited knowledge of Latin, German, and Japanese.) The needs of direct manipulation also fit within this model since we can describe anything we do verbally, although the actual interaction is qualitatively different.

A command is directed to the agent that is to perform the action. In the real world this is done by naming the agent or by getting the agent's attention and addressing him/her directly. The latter is what we do when we select a window. The former depends on a naming scheme which Pink will probably address via the 'forms' mechanism described in the strawman section above.

The next part is the naming of the action to be performed. Pink uses the old stand-bys of menus, buttons, and controls to do this and also provides a textual form that is used in scripts and message boxes (if the latter is supported).

There may be a subject of the action, e.g. the document to be discarded. This is the roll played by the selection, if any.

Finally, one or more phrases may be included that specialize the action to this situation. Direct manipulation interactions may allow for the equivalent of a single phrase, e.g. the destination of a

move. Complex phrase structures, e.g. printing instructions, are represented by dialog boxes. Dialog boxes can be tedious, specially when the defaults are correct, so Pink is considering mechanisms that support 'prepackaged' command-dialog sets.

Interaction

Interactions are the way a person and machine communicate. This section deals with the syntax of that communication, not the content. The syntax of the content was dealt with in the previous section; the symantics of the content are contained in the Taxonomy section below.

People interact with the world, including people, in two distinct ways: symbolically and directly. Symbols are representations of something else, and each symbol's meaning must be agreed upon by its users. Written and spoken languages are symbolic. Direct interaction deals with way people manipulate tangible objects in the physical world, things that we can touch, see, smell, and hear.

Pink inherits from Lisa/Macintosh direct manipulation, manually-initiated symbolic interaction (menu selection and buttons) and, more recently, typed-in symbolic interaction via HyperCard and AppleScript.

A very good understanding of the way people interact both symbolically and directly with the world around them is essential to Pink. A poor understanding will likely result in an interface that is, at best, irritating, slow, tedious, etc. A companion paper on the every-user model is in preparation.

Symbolic

The model of symbolic interaction is verbal/written language which was briefly described in the Operations section above. Pink will support symbolic interaction, both directly via menus and buttons and textually in the form of scripts. [Extensive work to be done in this section.]

Direct Manipulation

The most important factor affecting the quality of direct manipulation in Pink is the physical similarity of manipulations in Pink to their real world counterparts. Physical similarity is accomplished through the matching of peripheral hardware capabilities to those of the human channels they interact with and the use of similar manipulations to do the same thing in both Pink and the real world.

Caveat: Many real world controls are poorly designed or mismatched to the function, e.g. the use of 'up' and 'down' buttons to control volume. The existence of a control in the real world is not a priori justification for its use in Pink.

In the real world the hands (plural) and fingers are used:

- to point out objects of interest,
- to grab objects,
- to position and rotate objects in six ways,
- to sense an object's temperature, texture, shape, and size, and
- to apply varying amounts of pressure with different fingers.

When incorporating direct manipulation into Pink, it is best to maintain a direct correspondence of action between the real world and Pink's virtual world. For example, moving something in Pink should always be done by the user moving the mouse, rolling a track ball, or other comparable physical movement. The limited capabilities of the hardware, e.g the mouse, may make the implementation of comparable actions impossible and require the creation of artifacts in order to support a needed operation.

Design Language

Elements of the Pink design language should look plausible, as if they could exist in the real world. Elements are represented with "cartoons" or caricatures that indicate function through form. On the Macintosh and Lisa, buttons had rounded edges and clearly delineated regions and were clicked. Menus were square with a shadow and contained more than one item and were pulled down. Icons were cartoons with a variety of shapes but one particular size and could be dragged. Icons were not drawn as buttons, nor buttons drawn as menus, and so on.

This applies to the visual behavior of the elements as well; if the element could exist, then it should behave as that real world element might behave. If a control in the real world flattens when pressed, then the counterpart in the Pink model should also appear to flatten. On the Macintosh, the model for controls is based upon highlighting them when the user activates them.

Elements should not only look plausible, but they should provide clues to the user as to their function. If an element is to be interacted with in a particular fashion, say dragged versus clicked, then the appearance of the element should subtlety indicate this. Indeed, all elements that share a common interaction should provide a similar consistent set of visual clues. On the Macintosh the graphic indication of a grow box hints at its behavior. On Pink the appearance will also hint at the kind of behavior the element requires. Keep in mind that a successful element is not judged by its first impression but by its successful use after two or three tries on a user's part and its successful reuse after a significant intervening period of time.

All elements the user interacts with must be visible. On the Macintosh, which has a flat 2D physical model, controls have an outline or frame that delineates them from the background and a consistent representation. In the Pink 2 1/2D physical model, controls in a dialog rest on top of the flat area of the block and have a consistent representation.

Given the use of animation for transition, state/progress indication, and realistic operation, animated elements will behave plausibly and provide clues as to the action taking place. If the animation represents a transition, like open, than it should be smooth and unobtrusive, completing the illusion of the model and only noticed if it did not happen. If the animation represents a state/progress indicator, then it should show progress evenly without distraction. Animations of realistic operation should give the user clues as to the kind of operation occurring; for example, is the animation representing a copy or a recalculation? Animations of realistic operations should also be consistent within the kind of element that they are occurring; if an icon is animated versus a window, then the animation is appropriate to the element.

Color is used to represent the illusion of a three dimensional model fully. In addition, color is another redundant clue that helps to clearly indicate edges, regions and function. The interior of a grow element may be subtlety distinct from a button element. The edge an alert element may have a color, in addition to the alert icon, to indicate the kind of alert. The edges of control and the edges of a window will have different contrast to indicate that the control is contained within the window.

Physical Model

The physical model the Pink look is patterned after is a thin three dimensional block, 1/16" to 1/8" thick, that has rolled or extruded edges. The rolled edges are smooth with no abrupt planes. The surface of the block is flat. To show content one exposes the interior of the block. To control the block, one puts controls on top of the block. This is similar to a TV. One can watch the content which appears on the inside of the TV set, or control the content with the dials and knobs on the edges of the TV set frame.

The light source on the block is from the front, approximately centered on the user's nose. This provides for even illumination. Shadows, if any, are transparent.

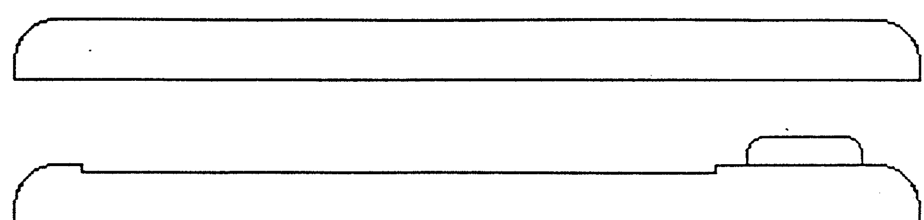


Figure 9. Side view Pink interface objects

Taxonomy

This section describes all of the generic kinds of objects, operations, and interactions. The large size of this taxonomy is of great concern since even inexperienced users will come into contact with much of it. It a large number of items and so has been organized in a way that a user might consciously group them. Other organizations are possible and maybe even arguably better.

Objects

This section organizes Pink objects into seven groups as a user might. (A limited user survey designed to elicit typical organizations is being prepared.) Many of the objects exist in the Lisa and/or Macintosh (Blue) and are therefore mentioned only briefly. Those that are unique to Pink are explained in more detail.

People

In some situations, both home and work, several individuals will use a machine at different times. People objects represent the individuals and groups that have access to the machine both directly or through the network. The properties of a person object include individual information, e.g. name and picture, preferences for the system and applications, and access rights to things on the machine (but not other machines). Applications should store their user preferences here instead of in the System Folder.

By default, the system contains at least one visible person object. If several people share the machine, each should have a corresponding person object. The system will use the current person object for preferences, identification, and access control. In the future it might contain voice recognition patterns or anything else that is user specific.

Person objects are used to control remote access. If a person object exists for a remote user, he has access as determined by the content of his person object.

Environment

The computer object represents the CPU, all attached peripherals including networks (but not the objects on the network), and all system software including the Finder, drivers, cdevs, and inits,(but not applications or system utilities). It replaces the Blue system folder and control panel. It provides a graphical means and indication of what is in the machine (including memory size and boards), configuration information, what is connected to what and mechanisms for holding and activating both system software and system level augmentations. The computer object from the start up disk appears on the desktop. Computer objects on other disks appear in the disk window; they cannot be placed in folders.

Other environmental objects, e.g. Fonts and Help files, also reside within the computer object.

Document-based applications could also be included here, maybe.

One network object, currently the 'phone book', appears on the desktop to represent all attached networks and network objects. References to network objects can be dragged from the phonebook to the desktop, folders, etc. to provide quick access at later times. The network object cannot be dragged onto/into a disk, folder, or other container.

Each peripheral attached to the machine or network, e.g. disks, scanners, monitors, etc., is represented by an object. These objects are normally found within the computer or network objects as appropriate. These objects can be opened to set configuration and user preference information. (User preference info is actually stored with the current person object.) Peripheral objects can be moved to the desktop, leaving a gray place holder behind. The peripherals include: disks, diskette drives, keyboard, mouse, monitor(s), scanner, printers, other computers, telephones, etc.

The desktop is essentially the same as the Mac desktop except that it remembers the placement of all objects on it across reboots.

Data

This groups contains the information objects that relate directly to the user's tasks. There are three main subdivisions, the basic data objects themselves, the structures that relate them to each other, and the marking tools used to create and edit them.

The basic data objects include text (characters, words), graphics/images (bitmaps, shapes, pixels, overlays), and sound(samples, notes).

The structures that relate the basic data objects include linear (lists, tables, arrays), parallel (time and event synchronization), nested (outlines, containers), branching (trees, graphs), and spatial (2D, 2 1/2D, 3D).

The marking tools include the pen, brush, keyboard, and spray can.

Tools

Tools are the behavioral objects that a user consciously interacts with as opposed to applications and system software which the user need not see or deal with except when installing/updating it. This group includes appliances (whole document mungers), tools (non-document things like calculators), and scripts. See the Thor section, 3.1.1, for a more extensive discussion of these objects.

Connections/Navigation aids

This group contains objects that mainly serve to relate otherwise disjoint objects regardless of the purpose. They include navigation (links, references, maps), communication (phonebook), and dataflow (warm/hot links).

Containers

In this group are the objects that hold data and the objects that hold the data holders. The data holders include documents (documents, drawings, images, stacks, spreadsheets, calendars, notebooks, scrapbooks, clipboards), annotations, recordings (sounds, animations, video clips, movies, slide shows), multi-media (may have several physical pieces), stationery, forms, and simulations (maybe this is just an active document). The holder holders include the desktop, folders, trash can, mailbox, disks (double grouped?)

Interaction Controls

This last group contains all the objects used for (in)direct manipulation of Pink's virtual reality. They are all artifacts, i.e. objects that generally do not exist in the real world but which are required by the

computer to overcome hardware limitations. Ideally they would exist only in places where comparable real world objects exist or where Pink has extended the real world metaphor. Included are windows (windows, dialogs, alerts, utility windows, subwindows, window sets, progress indicators), menus (menu bar, pull-down, pop-up, tear-off, menu title), buttons (single action, continuous action, check box, radio, command keys, cursor keys), window parts (scrollbars, grow box, zoom box, close box, window splitters), virtual track ball, etc.

Operations

Regardless of the actual tasks a person performs, all actions can be classified into one of the following categories:

- Observe — gather information through the five senses
- Navigate — ascertain current location, navigate in real and symbolic space (of self)
- Ideate — subconscious generation of ideas, alternatives
- Decide — chose one or some from several
- Select — to indicate which one(s)of several
- Manipulate — reason, organize, build, orient objects (other than self)
- Communicate — share with others

(We ignore essentially physical activities like running or throwing a spear since they don't seem to be applicable to computers.)

These categories are used to group the operations provided by computing systems.

Observe	open, preview, set aside, close, put away, run, interrupt, resume, stop, restart, magnify, change view
Navigate	position, scan, step, skip, jump to, traverse, where am I,
Ideate	(no operations appear to fit within this category)
Decide	(no operations appear to fit within this category)
Select	select one, many, all, adjust, filter
Manipulate	create, destroy, initialize, clear, copy, compare, cut, paste, replace, merge, insert, repair, verify, size, shape, color, move, align, rotate, flip, connect, group, disconnect, ungroup, undo, redo, repeat, checkpoint, revert, save version
Communicate	send, get, connect to, disconnect, address

Interaction

Regardless of the actual operation being performed there are three aspects to interaction, the person-to-machine side (direct and symbolic), and the machine-to-person side (feedback).

Direct	Move, rotate, press, release, click, double click, drag (press, move, release), (press, hold, release)
Symbolic	click-on command, type-in command, dialog, spoken command
Feedback	static display, animation, sound, visual change, completed list

The Pink Run Time System

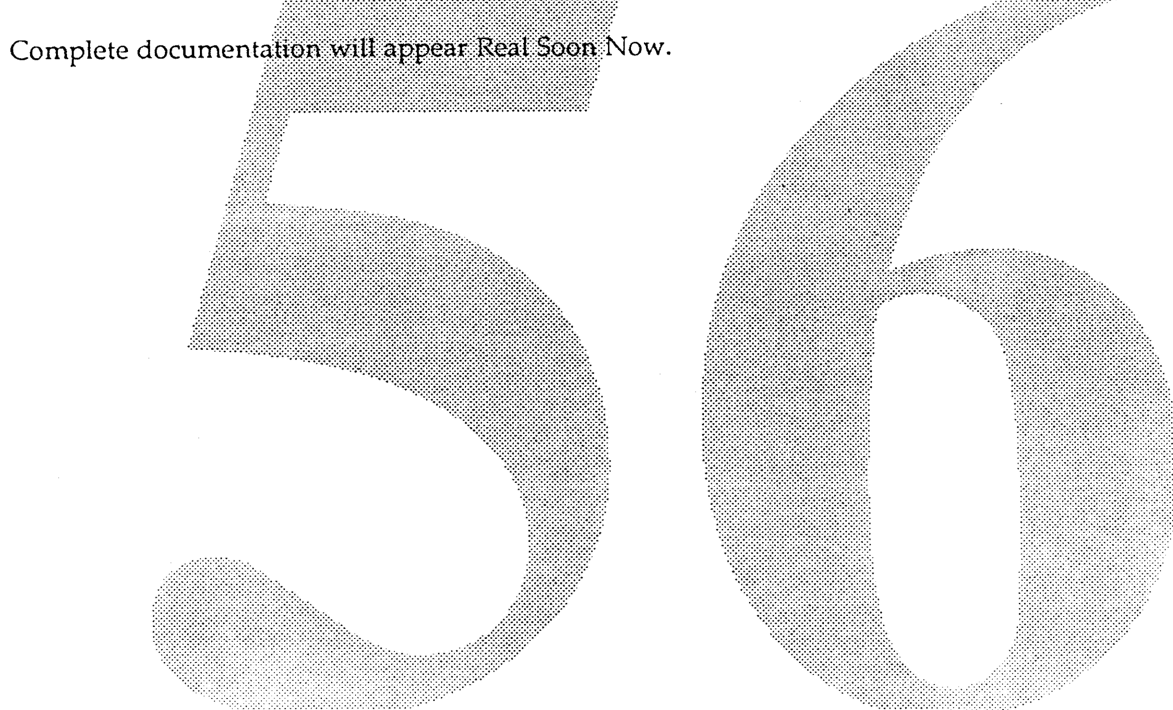
56

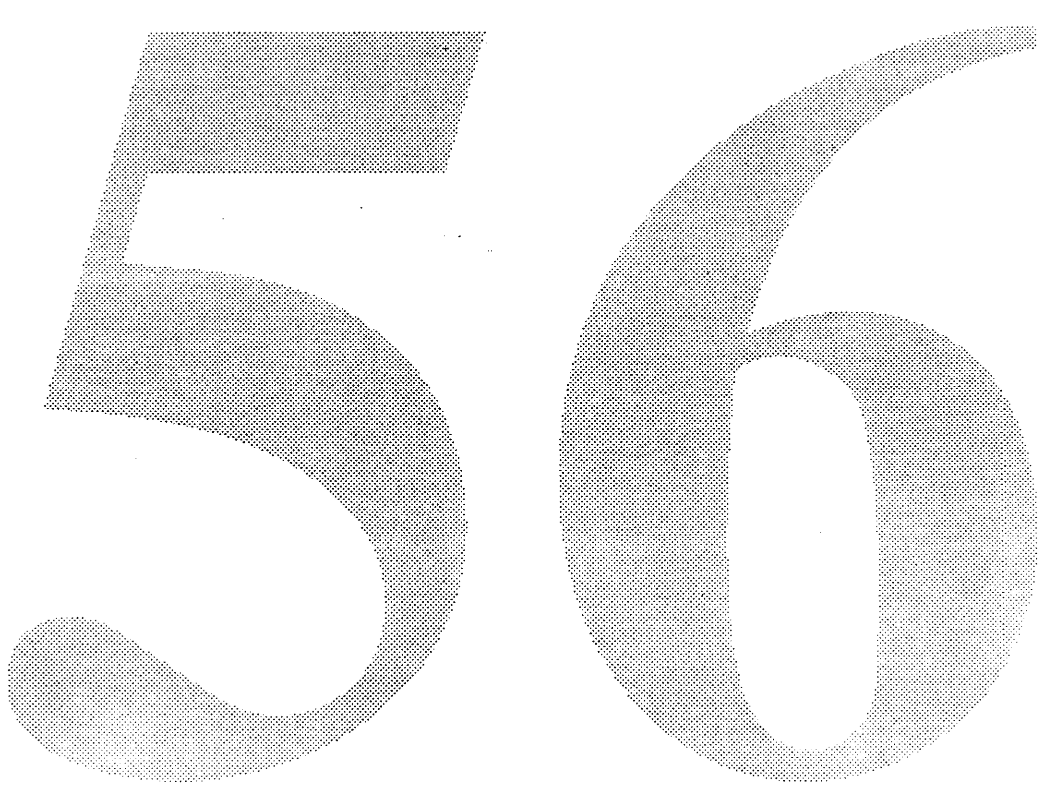
Architecture

The Pink Run Time system will provide Pink with these capabilities

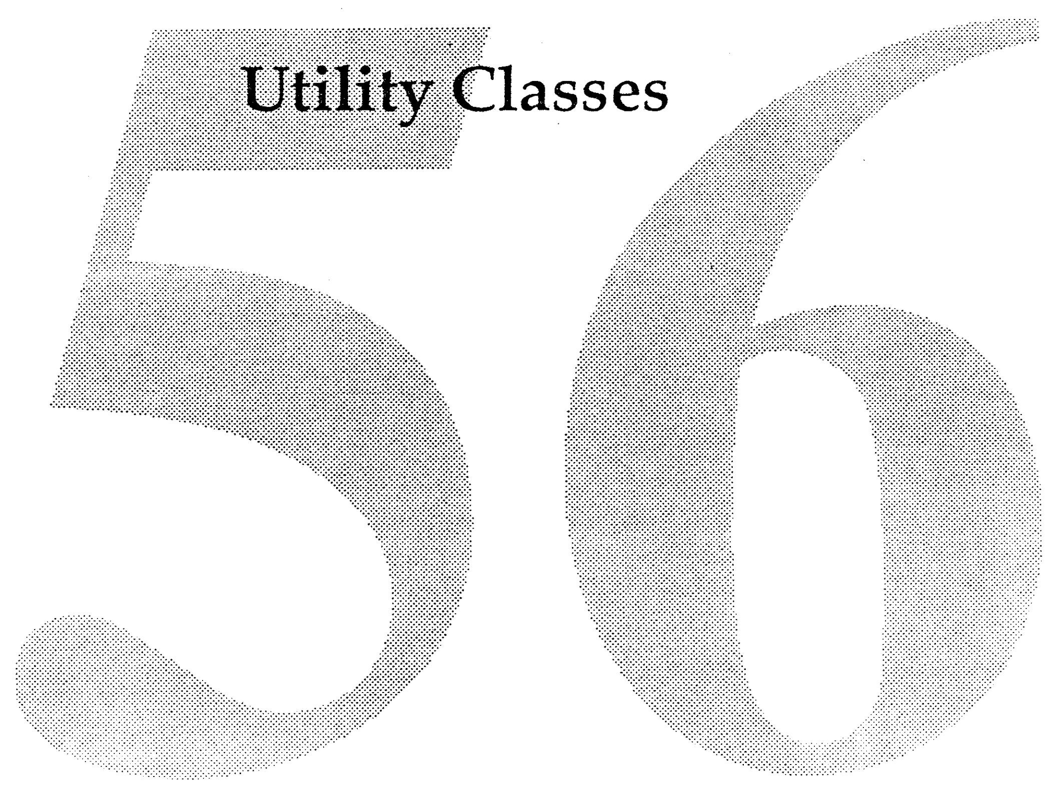
- Shared Libraries of code, providing for small applications, and for updating the system without breaking applications.
- A storage allocator.
- Semaphores, for use in synchronizing multi-threaded applications.
- Exception handling, provided in conjunction with the C++ language.
- Libraries of classes that may be dynamically accessed during program execution.

Complete documentation will appear Real Soon Now.





Utility Classes



56

Utility Classes

56

Arnold Schaeffer x8117

I hate data structures. You probably do too. Fortunately for you, Pink provides a set of classes for dealing with the most common data structures you are likely to need. Use these classes whenever you can because your code will get faster as we make performance enhancements to these "utility" classes.

56

Introduction

I hate data structures. You probably do too. Fortunately for you, Pink provides a set of classes for dealing with the most common data structures you are likely to need. Use these classes whenever you can because your code will get faster as we make performance enhancements to these "utility" classes.

The Utility Classes are roughly divided into two sections: the Collection classes and the CS101 classes. The Collection classes provide a set of classes somewhat equivalent to the collection classes found in Smalltalk. These classes include sets, bags, dictionaries, stacks, deques, queues, priority queues, dynamic Arrays, sorted sequences and run arrays. The Collection classes are implemented using the CS101 classes which are "raw" data structure classes like hash tables, linked lists, heaps, trees, etc. Most users of the Utility Classes should only use the Collection classes. The choice of which collection class to use allows you to specify the kind of operations you expect to do on a data set as well as some hints as to the expected size of the data set. The computer can then choose the proper CS101 class to use as an implementation.

Architectural Overview

The Utility Classes are basically a set of classes for managing and manipulating data. The particular utility class that you use will be determined by the kind of operations that you would like to perform on the data. For the purposes of this architectural overview, it is assumed that the utility classes you will be using are the collection classes. Other classes contained in the utility classes are used only as implementations and, in most cases, shouldn't be directly used.

All of the collection classes support a common protocol at the base level. Subclasses of the baseclasses add protocol for spec ialized behavior. The base class of all collections (TCollection) provides methods for **adding** elements to the collection; **removing** elements from the collection; **querying** whether an object is an element of a collection; **counting** the elements in a collection; **removing all** the elements from a collection; **adding all** of the elements from one collection into another collection; **destroying** (and removing) all of the elements from a collection; and **enumerating** over the elements of a collection.

Deciding what subclass of TCollection you should use depends on the additional operations you are performing on the collection. If it is important that order is preserved in a collection, then a collection that has a notion of order should be chosen. Collections such as deques, stacks and queues all maintain the order of the elements put into them. Stacks have a policy that the last element added to it is the first element removed. Queues have a policy that the first element added is the first element removed. Stacks and queues are good choices when the data you are managing follows one of these policies. Deques are ordered (like stacks and queues) but there is no implicit element removing policy; therefore, elements can be added at any place in the deque and removed in any order. Deques, stacks and queues are all ordered by some external (to the elements) procedure. The individual elements in these collections have no internal notion of order. Operations on stacks, queues, and deques that involve adding/removing from the beginning or the end are all O(1). Operations on stacks, queues, or deques for arbitrary removing/querying are all O(N).

If the elements of the collection have an external notion of ordering based on an index then there are two collections available for use. Dynamic Arrays allow elements to be associated with an index. Adding/Retrieving elements to/from the collection at a specific index is O(1). Growing the array is very expensive. Run Arrays allow elements to be associated with an index. Run Arrays are very efficient when there are long runs of the same elements at contiguous index values. All operations on run arrays are O(logN).

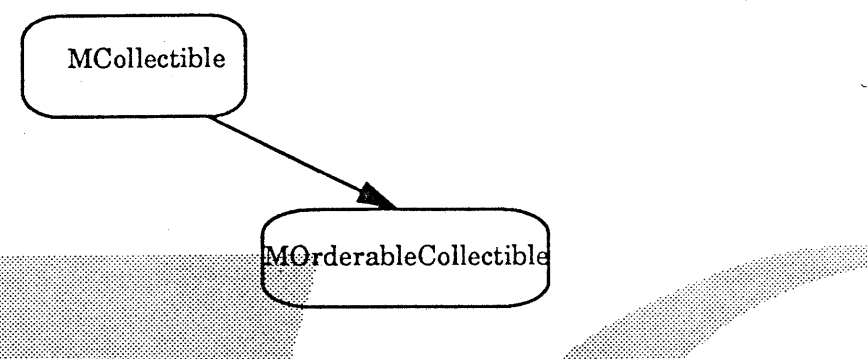
If the elements have an internal notion of ordering then there are two choices of collections that you can use. Priority Queues provide a collection where the elements are only partially ordered based on some internal notion of ordering. For example, many applications require that elements are processed in some order; however, not necessarily all at once or in fully sorted order. Examples of this kind of collection could be found in an event scheduling system where the most urgent element is always scheduled first. SortedSequences provide a collection where the elements are fully ordered based on some internal notion of ordering. Of course, operations on sorted sequences are more expensive than operations on a priority queue. This is because there is some overhead to pay for maintaining the sorted sequence. Operations on priority queues are all O(logN). Operations on Sorted Sequences are also O(logN); however, there is significant overhead in the balancing mechanism. Sorted Sequences are optimized for access speed at the expense of update speed (i.e. it is assumed that access happens more frequently than add or remove).

Unordered collections have no notion of ordering. Iterating through an unordered collection returns the elements of the collection in a random order. All operations on unordered collections are O(1). Unordered collections include sets, bags and dictionaries. Sets are a collection of elements. Sets support the additional protocol of union, intersection, xor, and difference. Dictionaries (associative arrays) associate an element designated as the key with an element designated as the value. Adding to a dictionary involves supplying a key, value pair to thedictionary. Retrieval of a value is possible given a key.

Naturally, all of this stuff doesn't come for free. In order to accomplish this magic, elements which are collected must mixin a class which defines protocol used by the collection classes. Methods must be overridden for providing comparison function, ordering functions and hashing functions appropriate to the particular subclass.

Generic Objects

The class `MCollectible` defines the generic object class from which all other classes are derived. It is an abstract class in that many subclasses will redefine some or all of the methods presented below. `MOrderableCollectible` should be mixed into objects which might have to be ordered. If you wish to use the utility classes, your objects should descend from one of these classes.



MCollectible

The class MCollectible defines the generic object class from which all other classes are derived. It is an abstract class in that many subclasses will redefine some or all of the methods presented below. There are also methods in MCollectible for streaming objects. These methods are described in the Cheetah spec. Subclasses should include the line:

```
MCollectibleDeclarationsMacro(subclassName);
```

in their declaration (.h) and should include the line:

```
MCollectibleDefinitionsMacro(subclassName);
```

in their definition file (.c).

```
class MCollectible {
  public:
    MCollectible();
    virtual ~MCollectible();
    virtual long            Hash();1
    virtual boolean         IsEqual(const MCollectible*  obj);
    virtual boolean         IsSame(const MCollectible* obj);
    virtual MCollectible*   Clone() const;
    inline  boolean         operator==(const MOrderableCollectible& obj);
    inline  boolean         operator!=(const MOrderableCollectible& obj);
  }

typedef boolean (MCollectible::* MCollectibleCompareFn)(const MCollectible*);
typedef long (MCollectible::* MCollectibleHashFn)();
```

`boolean MCollectible::IsSame(const MCollectible* obj)`
The default function is a pointer comparison.

`long MCollectible::Hash()`
Returns a value suitable for use as a hashing probe for this. The default function will simply return the address of the object. The default function is almost certainly not adequate if you are overriding `IsEqual` because you need to make sure that all objects that "are equal" to each other return the

1. Bold type is used for methods which almost always should be overridden.

same hash value. For example, a TText object would return a hash value computed using the characters in the string instead of the address of the string.

boolean MCollectible::IsEqual(const MCollectible* obj)
Returns TRUE if obj is isomorphic to this. The default function will throw you into OpusBug and give you a nasty message. For example, the IsEqual method for TText objects will do a string comparison. All of the utility classes allow you to specify what method to use when comparing objects for insertion, deletion, etc.

MCollectible* MCollectible::Clone() const
This method is declared and defined automatically when using the MCollectibleDeclarationsMacro. It is always defined as { return new subclassName(*this); }. This provides a general polymorphic duplication function.

MOrderableCollectible

MOrderableCollectible should be mixed into objects which might want to be ordered. Objects which are passed to TPriorityQueues, TSortedSequence, TTrees, etc. must have MOrderableCollectible mixed into them.

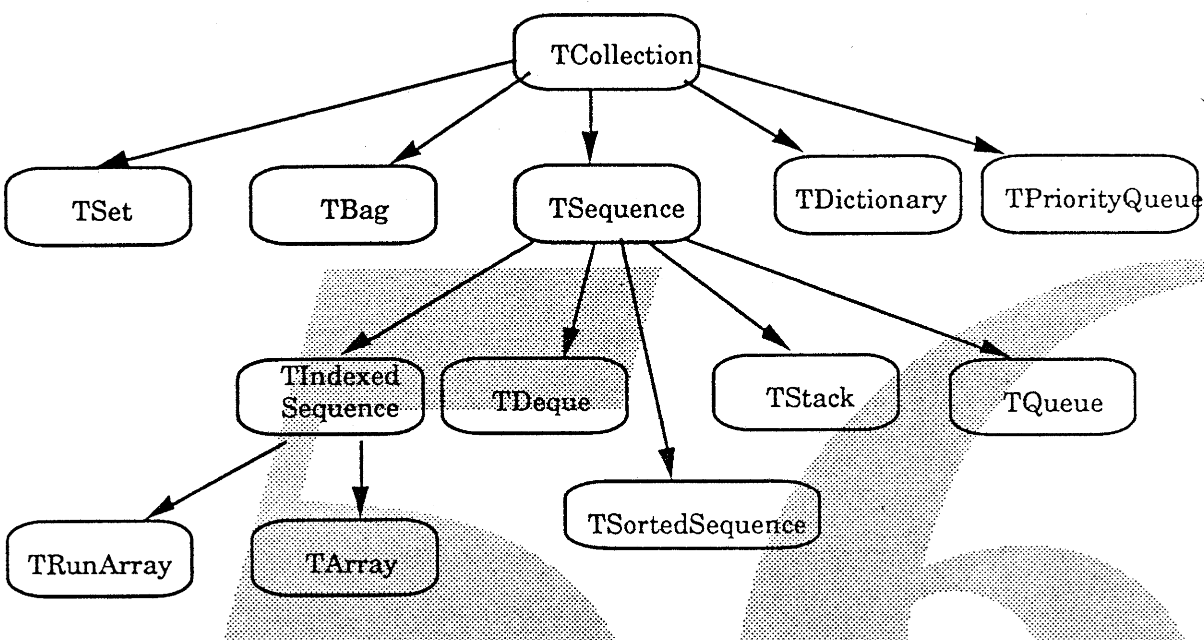
```
class MOrderableCollectible : public MCollectible{
  public:
    MOrderableCollectible();
    virtual ~MOrderableCollectible();
    virtual boolean IsGreaterThan(const MOrderableCollectible* obj);
    virtual boolean IsLessThan(const MOrderableCollectible* obj);
    inline  boolean  operator<(const MOrderableCollectible& obj);
    inline  boolean  operator>(const MOrderableCollectible& obj);
    inline  boolean  operator>=(const MOrderableCollectible& obj);
    inline  boolean  operator<=(const MOrderableCollectible& obj);
  }

typedef boolean (MOrderableCollectible::* MOrderableCollectibleCompareFn)
         (const MCollectible*);
```

boolean MCollectible::IsGreaterThan(const MOrderableCollectible* obj)
Returns TRUE if obj is "greater than" this. The default function will throw you into OpusBug and give you a nasty message. For example, the IsGreaterThan method for TText objects will do a string comparison.

boolean MCollectible::IsLessThan(const MOrderableCollectible* obj)
Returns TRUE if obj is "less than" this. The default function will throw you into OpusBug and give you a nasty message. For example, the IsLessThan method for TText objects will do a string comparison.

Collection Classes

Collection classes are used to group objects in meaningful ways. The system provides implementations of many of the collection classes from Smalltalk. Use these classes. They will get faster, smaller, taste better, and less filling.



TCollection

A `TCollection` represents a group of objects. It is implemented as an abstract class from which all collection classes inherit methods. There are no instances of `TCollection`, only subclasses. Collections all provide a facility for iterating over their members. This facility is described in detail in the section on iterators (see page 35).

```
class TCollection: public MCollectible {

  public:
    TCollection(MCollectibleCompareFn testfn);
    virtual ~TCollection();
    virtual void            Add(MCollectible* obj);
    virtual void            Add(TCollection* collection);
    virtual MCollectible*   Remove(const MCollectible& obj);
    virtual void            RemoveAll();
    virtual void            DeleteAll();
    virtual long            Count() const;
    virtual MCollectible*   Member(const MCollectible& obj) const;
    virtual TIterator*      CreateIterator() const;

}
```

`TCollection(MCollectibleCompareFn testfn)`
Create a new `TCollection`. All future operations will use `testfn` for a comparison when needed.

`virtual ~TCollection()`
Destroy the mother.

`void TCollection::Add(MCollectible* obj)`
Add obj to this.

`void TCollection::Add(TCollection* collection)`
Add all of the objects in collection to this. Essentially equivalent to getting an iterator for the collection passed in and adding each element in the `collection` to `this`.

`MCollectible* TCollection::Remove(const MCollectible& obj)`
Remove `obj` from `this`. Return the object which was actually removed (which if you are using an `IsEqual` test function may not be the same as the object passed in only "equal.")

`void TCollection::RemoveAll()`
Remove all of the objects from `this`.

`void TCollection::DeleteAll()`
Remove all of the objects from this and deallocate the storage that these objects might have owned (that is, the destructor function is called for each object in the collection).

`long TCollection::Count() const`
Return the number of objects in `this`.

`MCollectible* TCollection::Member(const MCollectible& obj) const`
Each object in this is compared to obj using the function testFn. Return the object for which testFn returned true. Return `NIL` if no object was found.

`TIterator* TCollection::CreateIterator() const`
This method returns a new iterator which is suitable for use in iterating over the objects in `this` collection. See the special section on iterators on page 35.

TBag

The `TBag` class is a subclass of `TCollection`. It represents an unordered collection of objects in which objects can appear more than once. Objects which are inserted into the `TBag` should override the `Hash()` method and the `IsSame()` or `IsEqual()` method.[2]

```
const long kInitialBagSize;

class TBag: public TCollection {
   public:
     TBag(MCollectibleCompareFn testFn = &MCollectible::IsSame,
          long bagSizeGuess=kInitialBagSize);
    virtual                      ~TBag();
    virtual void           AddWithOccurrences(MCollectible* obj, long number);
    virtual long           OccurrencesOf(const MCollectible* obj) const;
    virtual MCollectibleHashFn  GetHashFunction(MCollectibleHashFn) const;
```

2. If you are using an `IsEqual` `TBag`, only the first object added that "is equal" to other objects added is retained by the collection and returned as the result of calls to member, remove, etc. This can make memory management awkward. Think about this when using an "is equal" bag.

```
  virtual void                SetHashFunction(MCollectibleHashFn);
}
```

`TBag::TBag(MCollectibleCompareFn testFn, long bagSizeGuess);`
Create a bag which can hold at least bagSizeGuess elements. BagSizeGuess is used to determine what implementation class to use for bag.

`void TBag::AddWithOccurrences(MCollectible* obj, long number);`
Add obj to this with number of occurences number.

`long TBag::OccurrencesOf(const MCollectible* obj) const;`
Return the number of occurences of obj in this. Zero indicates that obj is not in the bag.

`MCollectibleHashFn TBag::GetHashFunction() const`
Return the hash function being used by the hash table.

`void TBag::SetHashFunction(MCollectibleHashFn)`
Set which member function to call as a hash function. By default, this is set to `&MCollectible::Hash` (which is usually overridden in the objects you are adding to the hash table). You can use any hash function that you like as long as it has the type signature of `MCollectibleHashFn` (which is basically a method taking no parameters and returning a long). Most of the time, you won't need to do this.

TSet

The `TSet` class is a subclass of `TCollection`. It represents an unordered collection of objects in which objects can appear only once. Objects which are inserted into the `TSet` should override the `Hash()` method and the `IsSame()` or `IsEqual()` method.

```
const long kInitialSetSize;

class TSet: public TCollection {
 public:
    TSet(MCollectibleCompareFn testFn = &MCollectible::IsEqual,
         long setSizeGuess=kInitialSetSize);
    virtual                     ~TSet();
    virtual void                Difference(const TSet& set1);
    virtual void                Difference(const TSet& set1, TSet& result);
    virtual void                Intersection(const TSet& set1);
    virtual void                Intersection(const TSet& set1, TSet& result);
    virtual void                Union(const TSet& set1);
    virtual void                Union(const TSet& set1, TSet& result);
    virtual void                Xor(const TSet& set1);
    virtual void                Xor(const TSet& set1, TSet& result);
    virtual MCollectibleHashFn  GetHashFunction(MCollectibleHashFn) const;
    virtual void                SetHashFunction(MCollectibleHashFn);
}
```

`TSet::TSet(MCollectibleCompareFn testFn, long setSizeGuess);`
Create a set which can hold at least setSizeGuess elements. SetSizeGuess is used to determine what implementation class to use for set.

`void TSet::Difference(const TSet& set1);`
Destructively modify `this` to contain a set of elements of `this` that do not appear in `set1`.

`void TSet::Difference(const TSet& set1, TSet& result);`
After this function is called, `result` will contain a set of elements of `this` that do not appear in `set1`.

`void TSet::Intersection(const TSet& set1);`
Destructively modify `this` to contain everything that is an element of `set1` and `this`.

`void TSet::Intersection(const TSet& set1, TSet& result);`
After this function is called, `result` will contain everything that is an element of `set1` and `this`.

`void TSet::Union(const TSet& set1);`
Destructively modify this to contain everything that is an element of `set1` or `this`.

`void TSet::Union(const TSet& set1, TSet& result);`
After this function is called, `result` will contain everything that is an element of `set1` or `this`.

`void TSet::Xor(const TSet& set1);`
Destructively modify `this` to contain everything that is an element of either `set1` or `this`, but not both.

`void TSet::Xor(const TSet& set1, TSet& result);`
After this function is called, `result` will contain everything that is an element of either `set1` or `this`, but not both.

`MCollectibleHashFn TSet::GetHashFunction() const`
Return the hash function being used by the hash table.

`void TSet::SetHashFunction(MCollectibleHashFn)`
Set which member function to call as a hash function. By default, this is set to `&MCollectible::Hash` (which is usually overridden in the objects you are adding to the hash table). You can use any hash function that you like as long as it has the type signature of `MCollectibleHashFn` (which is basically a method taking no parameters and returning a long). Most of the time, you won't need to do this.

TDictionary

The class `TDictionary` is a subclass of `TCollection`. It represents a collection of paired objects (associations). Because dictionaries are sometimes used to represent a bijective mapping, functions for retrieving a key given a value are provided along with the usual access functions (however, this will probably be slow). Objects which are inserted into the `TDictionary` should override the `Hash()` method and the `IsSame()` or `IsEqual()` method. These are used internally by the `TDictionary` class. **Note: Iterators on the `TDictionary` class return objects of class `TAssoc`.**

```
const long kInitialDictionarySize;

class TDictionary: public TCollection {
 public:
   TDictionary(MCollectibleCompareFn testFn = &MCollectible::IsEqual,
```

```
                long dictionarySizeGuess=kInitialDictionarySize);
    virtual ~TDictionary();
    virtual MCollectible*         ValueAt(const MCollectible& key) const;
    virtual const MCollectible* KeyAt(const MCollectible& val) const;
    virtual MCollectible*         Remove(const MCollectible& key);
    virtual MCollectible*         DeleteKey(MCollectible* key);
    virtual void                  DeleteAllKeys();
    virtual void                  DeleteAllValues();
    virtual void                  AddKeyValuePair(const MCollectible* key,
                                          MCollectible* val,
                                          boolean replace = TRUE);
    virtual MCollectibleHashFn    GetHashFunction(MCollectibleHashFn) const;
    virtual void                  SetHashFunction(MCollectibleHashFn);
}
```

```
TDictionary::TDictionary(MCollectibleCompareFn testFn,
                         long dictionarySizeGuess);
```
Create a dictionary which can hold at least dictionarySizeGuess elements. DictionarySizeGuess is used to determine what implementation class to use for the dictionary.

```
MCollectible* TDictionary::ValueAt(const MCollectible& key) const
```
Return the value associated with the key. Return `NIL` if the key could not be found.

```
const MCollectible* TDictionary::KeyAt(const MCollectible& val) const
```
Return the first key found which has val as its value. This involves a slow search.

```
MCollectible* TDictionary::Remove(const MCollectible& key)
```
Remove the key, value pair associated with key. Return the value that was removed as a result of this call.

```
MCollectible* TDictionary::DeleteKey(MCollectible* key);
```
Delete the key from the key, value pair associated with key and remove the key,value pair from the dictionary. Return the value that was removed as a result of this call.

```
MCollectible* TDictionary::AddKeyValuePair(const MCollectible* key,
                                           MCollectible* val,
                                           boolean replace);
```
If replace=FALSE then only add key, value pair to the table if there is not an existing key,value pair. Otherwise, if replace=TRUE, add the key, value pair to the hash table. Either way, return the key that existed (if any) in the hash table before this call. Proper memory management may involve checking to see if the key returned is "the same" as the key passed in when replacing key, value pairs.

```
void TDictionary::DeleteAllKeys()
```
Remove all of the entries in the dictionary. Reset the count to be zero. Call the destructor on every key in the dictionary.

```
void TDictionary::DeleteAllValues()
```
Remove all of the entries in the dictionary. Reset the count to be zero. Call the destructor on every value in the hash table. If you have a value which appears more than once, you will be sorry you used this method because the utility classes will delete the same object more than once. This is not good.

```
MCollectibleHashFn TDictionary::GetHashFunction() const
```
Return the hash function being used by the hash table.

```
void TDictionary::SetHashFunction(MCollectibleHashFn)
```
Set which member function (of the objects in the dictionary) to call as a hash function. By default, this is set to `&MCollectible::Hash` (which is usually overridden in the objects you are adding to the hash table). You can use any hash function that you like as long as it has the type signature of `MCollectibleHashFn` (which is basically a method taking no parameters and returning a long). Most of the time, you won't need to do this.

TPriorityQueue

A `TPriorityQueue` is a subclass of `TCollection` which keeps the elements of the collection partially ordered based on some ordering function. Priority queues are often used when you must collect a set of records, then process the largest, then collect more, then process next largest, etc. There is considerable debate at this point as to whether this class really should be a subclass of `TCollection` since it relies on the good nature of the user to supply `MOrderableCollectibles` on the way in. Objects which are inserted into the TPriorityQueue should override the `IsEqual()`, `IsLessThan()` and `IsGreaterThan()` methods. These are used internally by the TPriorityQueue class. **Note: Iterators on a `TPriorityQueue` class do NOT return objects "in order." Because a `TPriorityQueue` is only partially ordered, this would be a very expensive operation in general. This functionality might eventually be supplied by the utility classes but not as the default behavior.**

```
class TPriorityQueue: public TCollection {
  public:
    TPriorityQueue(MOrderableCollectibleCompareFn  fn =
                        &MOrderableCollectible::IsLessThan);
    virtual ~TPriorityQueue();
    virtual void                   Insert(MOrderableCollectible* obj);
    virtual MOrderableCollectible* Pop();
    virtual MOrderableCollectible* Peek() const;
    virtual MOrderableCollectible* Replace(MOrderableCollectible* obj);
   virtual  MCollectibleCompareFn  GetEqualityComparisonFunction();
   virtual  void       SetEqualityComparisonFunction(MCollectibleCompareFn fn);
  }
```

```
TPriorityQueue::TPriorityQueue(MOrderableCollectibleCompareFn testFn);
```
Create a new priority queue. Use `testFn` to determine whether larger objects are removed first or last. A test of `IsLessThan` means that larger objects are removed first and smaller objects are removed last. A test of `IsGreaterThan` reverses this.

```
void TPriorityQueue::Insert(MOrderableCollectible* obj);
```
Insert obj in this and return it as a result.

```
MOrderableCollectible* TPriorityQueue::Pop();
```
Remove the object with the "highest" priority from the priority queue and return it.

```
MOrderableCollectible* TPriorityQueue::Peek() const;
```
Return the object with the "highest" priority from the priority queue but don't remove it.

`MOrderableCollectible* TPriorityQueue::Replace(MOrderableCollectible* obj);`
Roughly equivalent to inserting the obj into the priority queue and then removing the object with the highest prioriy.

`MCollectibleCompareFn TPriorityQueue::GetEqualityComparisonFunction() const`
Return the equality comparison function being used by the priority queue.

`void TPriorityQueue::SetEqualityComparisonFunction(MCollectibleCompareFn)`
Set which member function to call as the equality comparison function when removing objects from the queue, checking to see whether a given object is a member, etc. This defaults to `IsEqual`. Most of the time you won't want to change this.

TSequence

A `TSequence` is an abstract superclass for collections whose elements are ordered.

```
class TSequence: public TCollection {

  public:
    virtual  MCollectible*       After(const MCollectible& obj) const;
    virtual  MCollectible*       Before(const MCollectible& obj) const;
    virtual  MCollectible*       First() const;
    virtual  MCollectible*       Last() const;
    virtual  void                Concatenate(TSequence* col);
    virtual  long                OccurrencesOf(const MCollectible& obj) const;
    virtual  void                Reverse();
    virtual  TSequenceIterator*  SequenceIterator() const;
  }
```

`MCollectible* TSequence::After(const MCollectible& obj) const`
Return the object found after `obj` in `this` or `NIL` if `obj` is the last object in `this` or not found.

`MCollectible* TSequence::Before(const MCollectible& obj) const`
Return the object found before `obj` in `this` or `NIL` if `obj` is the first object in `this` or not found.

`MCollectible* TSequence::First() const`
Return the first object in `this`.

`MCollectible* TSequence::Last() const`
Return the last object in `this`.

`void TSequence::Concatenate(TSequence* aCollection)`
Concatenate `aCollection` onto the end of `this`.

`long TSequence::OccurrencesOf(const MCollectible& obj) const`
Return the number of times `obj` is in `this`.

`void TSequence::Reverse()`
`this` is destructively turned into a collection which contains the same elements as `this`, but with the order of the elements reversed.

56

TSequenceIterator* TCollection::SequenceIterator() const
This method returns a new sequence iterator which is suitable for use in iterating over the objects in the collection. Sequence iterators differ from normal iterators in that they can start at the last object or the first object and go in either direction. See the special section on iterators on page 35.

TDeque

A TDeque is a subclass of TSequence which is ordered based on the order elements are added to or removed from the collection. Objects which are inserted into the TDeque should override the IsSame() or IsEqual() method.

```
class TDeque: public TSequence {
 public:
   TDeque(MCollectibleCompareFn testFn = &MCollectible::IsSame);
   virtual ~TDeque();
   virtual void          AddLast(MCollectible* obj);
   virtual void          AddFirst(MCollectible* obj);
   virtual MCollectible* RemoveLast();
   virtual MCollectible* RemoveFirst();
   virtual void          AddAfter(const MCollectible& existobj,
                                  MCollectible* new);
   virtual void          AddBefore(const MCollectible& exist,
                                   MCollectible* new);
 }
```

TDeque::TDeque(MCollectibleCompareFn testFn);
Create a new TDeque.

MCollectible* TDeque::RemoveLast()
Remove the last object in this and return it as a result. Return NIL if the collection is empty.

MCollectible* TDeque::RemoveFirst()
Remove the first object in this and return it as a result. Return NIL if the collection is empty.

void TDeque::AddAfter(const MCollectible& existobj, MCollectible* new)
Add the new object after existobj in the collection.

void TDeque::AddBefore(const MCollectible& exist, MCollectible* new)
Add the new object before exist in the collection.

void TDeque::AddLast(MCollectible* obj)
Add obj as the last object in the collection.

void TDeque::AddFirst(MCollectible* obj)
Add obj as the first object in the collection.

TStack

A TStack is subclass of TSequence in which the last item added to the stack is the first item taken out of the stack (LIFO). Objects which are inserted into the TStack should override the IsSame() or IsEqual() method. The iterator for a stack will return objects in the order they would be

returned if repeated Pops were issued to the stack.

```
class TStack: public TSequence {
  public:
    TStack(MCollectibleCompareFn testFn = &MCollectible::IsSame);
    virtual ~TStack();
    virtual MCollectible* Pop();
    virtual void          Push(MCollectible* obj);
  }
```

`TStack::TStack(MCollectibleCompareFn testFn)`
Create a new stack.

`void TStack::Push(MCollectible* obj);`
Add the object to the top of the stack.

`MCollectible* TStack::Pop();`
Remove the object on the top of the stack and return it. Return NIL if nothing is on the stack.

TQueue

A TQueue is a subclass of TSequence in which the first item added to the queue is the first item taken out of the queue (FIFO). Objects which are inserted into the TQueue should override the IsSame() or IsEqual() method. The iterator for a stack will return objects in the order they would be returned if repeated Pops were issued to the stack.

```
class TQueue: public TSequence {
  public:
    TQueue(MCollectibleCompareFn testFn = &MCollectible::IsSame);
    virtual ~TQueue();
    virtual void          Insert(MCollectible* obj);
    virtual MCollectible* Pop();
  }
```

`TQueue::TQueue(MCollectibleCompareFn testFn);`
Create a new queue.

`void TQueue::Insert(MCollectible* obj);`
Add an object to the queue.

`MCollectible* TQueue::Pop();`
Remove the oldest object in the queue (*F*irst *I*n, *F*irst *O*ut). Return NIL if nothing is in the queue.

TSortedSequence

A TSortedSequence is a subclass of TSequence in which all of the objects in the collection always remain sorted. New objects will be inserted in sorted order. All objects in the sequence must be MOrderableCollectibles. There is considerable debate at this point as to whether this class really should be a subclass of TSequence since it relies on the good nature of the user to supply MOrderableCollectibles on the way in. Objects which are inserted into the TSortedSequence

should override the `IsEqual()`, `IsLessThan()` and `IsGreaterThan()` methods.

```
class TSortedSequence: public TSequence {
  public:
    TSortedSequence(MOrderableCompareFn testFn = &MCollectible::IsLessThan);
    virtual ~TSortedSequence();
  }
```

`TSortedSequence::TSortedSequence(MOrderableCompareFn testFn);`
Create a new sorted sequence.

TIndexedSequence

A `TIndexedSequence` is an abstract superclass for collections whose elements are ordered and can be randomly accessed via a numerical index.

```
class TIndexedSequence: public TSequence {

  public:
    virtual MCollectible*  Fill(MCollectible* obj);
    virtual long           LowBound() const;
    virtual long           HighBound() const;
    virtual MCollectible*  At(long index) const;
    virtual MCollectible*  AtPut(long index, MCollectible* obj);
    virtual boolean        Find(const MCollectible* obj, long& findresult,
                                  long start=0, long end=0);
    virtual MCollectible*  AtInsert(long index, MCollectible* obj);
    virtual void           Replace(TIndexedSequence* seq,
                                     TIndexedSequence* rep,
                                     long seqstart=0, long thisstart=0,
                                     longrepstart=0);
    virtual boolean        Search(TSequence* seq, long& searchresult,
                                    long seqstart=0, thisstart=0);
  }
```

`MCollectible* TIndexedSequence::Fill(MCollectible* obj);`
Fill `this` with elements equal to `obj`. This does not duplicate the object, just a pointer to the object.

`long TIndexedSequence::LowBound();`
Return the index of the lowest bound in this collection.

`long TIndexedSequence::HighBound();`
Return the index of the highest bound in this collection.

`MCollectible* TIndexedSequence::At(long index) const;`
Return the object in `this` at the `index`. If `index` is past the end of `this` then `FAIL`.

`MCollectible* TIndexedSequence::AtPut(long index, MCollectible* obj);`
Add the `obj` to `this` at the `index`. The object that was previously at this `index` is returned from AtPut. If `index` is past the end of the collection then `FAIL` and do not add this object to the collection.

```
boolean TIndexedSequence::Find(const MCollectible* obj, long& findresult,
                               long start=0, long end=0);
```
If there is an object in this which `IsSame` or `IsEqual` (depending on the test function for the indexed sequence) to `obj` then return `TRUE` and set findresult equal to the index of the object. Otherwise return `FALSE`. If `start` or `end` is specified (as longs), these are used to determine where in the collection to start and end searching.

```
MCollectible* TIndexedSequence::AtInsert(long index, MCollectible* obj);
```
Insert the `obj` into the `TIndexedSequence` at the specified index. Effectively, the indexed sequence is grown one object. If index is out of the bounds of the `TIndexedSequence` then `FAIL`.

```
void TIndexedSequence::Replace(TIndexedSequence* seq, TIndexedSequence* rep,
                               long seqstart=0, long thisstart=0,
                               long repstart=0);
```
Replace elements in this matching seq with elements in rep. Use seqstart, thisstart, and repstart to determine where in the source, destination and match string to start from.

```
boolean TIndexedSequence::Search(TIndexedSequence* seq, long& searchresult,
                                 long seqstart=0, thisstart=0);
```
Search for a subcollection of this which matches seq. If there is no such subcollection then the result of this function is FALSE. If a subcollection is found then return the index of the first object in the subcollection in searchresult and return TRUE as a result of this function. Seqstart and thisstart can be used to control where in each collection to start searching.

TArray

An `TArray` is an abstract subclass of `TIndexedSequence` which allows for random access of the elements of the sequence via a numerical index. Furthermore, arrays are guaranteed to provide constant access and update time. Objects which are inserted into the `TArray` should override the `IsSame()` or `IsEqual()` method. These are used internally by the `TArray` class.

```
class TArray: public TIndexedSequence {
 public:
   TArray(MCollectibleCompareFn testFn = &MCollectible::IsSame,
          long initialSize=1, long offset=0);
   virtual ~TArray();
   virtual void          Grow(long howmuch, long extraspace = 0,
                              Boolean addToTop = TRUE);
   virtual void          Compress(long from, long howmuch);
   virtual MCollectible* Append(MCollectible* obj);
   virtual void          GrowTo(long maxIndex);

   virtual void          SetAutoGrowFlag(Boolean autoGrow = TRUE);
   virtual Boolean       GetAutoGrowFlag() const;
}
```

```
TArray::TArray(MCollectibleCompareFn testFn,
               long initialSize, long offset=0)
```
Create a new array of size `initialSize` and fill it initially with `NIL`. The offset of the first element of the array is `offset`.

```
void TArray::Grow(long howmuch, long extraspace, Boolean addToTop)
```
Grow the indexed sequence by `howmuch`. The additional elements can be inserted at the top of the array or the bottom depending on the value of the `addToTop` flag. `extraspace` is the amount of extraspace to use as a slush fund for future `AtInsert` operations to avoid copying the whole array.

```
void TArray::GrowTo(long index)
```
Grow the array in whatever direction is necessary to make `index` a valid index into the array.

```
void TArray::Compress(long from, long howmany)
```
Compress (remove) entries from the array beginning at entry `from` and continuing for `howmany` entries.

```
MCollectible* TArray::Append(MCollectible* obj)
```
Shorthand for `AtInsert(HighBound() + 1, obj)`.

```
void TArray::SetAutoGrowFlag(Boolean autoGrow = TRUE)
```
Set the `autoGrowFlag`. If `autoGrowFlag` is `TRUE` then instead of `FAIL`ing when an array index is out of bounds, automatically "grow" the array to accomodate the index. Actually, only grow on inserts and puts. Returns `NIL` to any At operations past the array bounds (a "virtual" grow).

```
Boolean TArray::GetAutoGrowFlag()
```
Return the `AutoGrowFlag`.

TRunArray

A `TRunArray` is a subclass of `TIndexedSequence` which provides space-efficient storage of data that tends to be sparse over long runs. The `TRunArray` stores repeated elements as single elements with a count associated with each one. The public access to `TRunArrays` look like the accessors to other arrays. The implementation of `TRunArrays` provide for fast lookup of the object at a given index and fast insert of an object at a given index. All accesses and updates occur in logarithmic time. (There is a separate paper which describes the implementation of the run array. Please request a copy of the paper if you are interested in how this is done.) Objects which are inserted into the `TRunArray` should override the `IsSame() or IsEqual()` method. These are used internally by the `TRunArray` class.

```
class TRunArray: public TIndexedSequence{
 public:
   TRunArray(MCollectibleCompareFn testFn = &MCollectible::IsSame,
   long size=1, long offset =0);
   virtual ~TRunArray();
   virtual void          AtCountPurge(long index, long count);
   virtual MCollectible*   AtCountReplace(long index, long count,
                                 MCollectible* obj);
   virtual MCollectible*   AtCountInsert(long index, long count,
                                   MCollectible* objt);
}
```

```
TRunArray::TRunArray(MCollectibleCompareFn testFn,
                     long size, long offset = 0);
```
Create a new run array of size = 1 with offset zero for the first element.

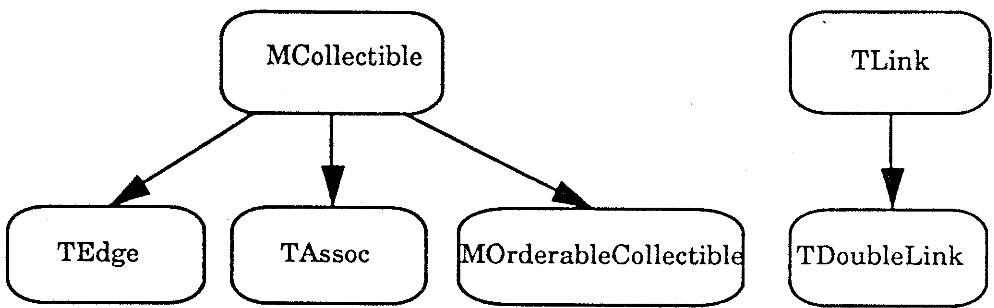
`MCollectible* TRunArray::AtCountInsert(long index, long count, MCollectible* obj);`
Insert a run of obj, count objects long into the run array at the specified index. Effectively, the array is grown by count objects. If index is out of the bounds of the array then return NIL.

`void TRunArray::AtCountPurge(long index, long count);`
Remove the objects in the run array starting at index and continuing until index+count-1. If either the upper or lower bounds of the element to be removed set is out of the array bounds then return NIL and do not perform the deletion.

`MCollectible* TRunArray::AtCountReplace(long index, long count, MCollectible* obj);`
Loosely equivalent to a call to `PurgeAt` to remove objects in the run array followed by an `InsertAt` to replace objects. Return NIL if the start or end of the replacement range is out of bounds. Otherwise return obj.

Simple Classes

These simple classes are used in the implementation of many CS101 classes.



TAssoc

A `TAssoc` is used to hold a pair of objects. Typically, these structures are owned by some other higher level object (e.g. a dictionary) and are usually not returned to the user. Users implementing their own classes might wish to use TAssocs in their implementations.

```
class TAssoc : public MCollectible {
  public:
    TAssoc();
    TAssoc(MCollectible* key = NIL, MCollectible* value = NIL);
    virtual                 ~TAssoc();
            void            SetKey(MCollectible* key);
            MCollectible*   GetKey();
            void            SetValue(MCollectible* value);
            MCollectible*   GetValue();
  }
```

TLink

A `TLink` is primarily used in the implementation of linked lists. Typically, these structures are owned by some other higher level object (e.g. a dictionary) and are usually not returned to the user. Users implementing their own classes might wish to use `TLinks` in their implementations.

```
class  TLink  {
  public:
    TLink();
    TLink(TLink* link = NIL, MCollectible* obj = NIL);
    virtual                 ~TLink();
    virtual  void           SetNext(TLink* link);
    virtual  TLink*         GetNext();
    virtual  void           SetValue(MCollectible* obj);
    virtual  MCollectible*  GetValue();
  }
```

TDoubleLink

A TDoubleLink is primarily used in the implementation of doubly linked lists. Typically, these structures are owned by some other higher level object (e.g. a dictionary) and are usually not returned to the user. Users implementing their own classes might wish to use TDoubleLinks in their implementations.

```
class  TDoubleLink : public TLink  {
  public:
    TDoubleLink();
    TDoubleLink(TLink* link = NIL, MCollectible* obj = NIL);
    virtual               ~TDoubleLink();
    virtual  void          SetPrevious(TLink* link);
    virtual  TDoubleLink*  GetPrevious();
  }
```

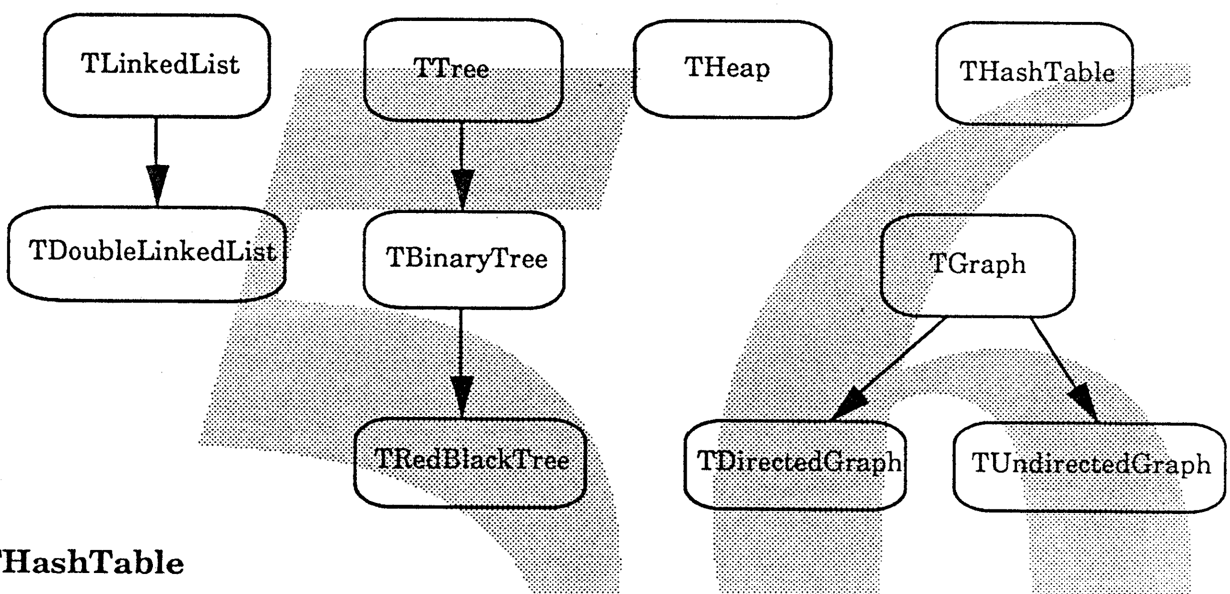
TEdge

A TEdge is an object used to communicate information about edges in a graph between the user and the system. It can be thought of as a copy of some aspect of a graph; however, it is not part of the graph itself because graph objects own their representations.

```
class TEdge : public MCollectible {
  public:
    TEdge();
    virtual ~TEdge();
    TEdge(TVertex* from, TVertex* to, WeightType weight=1);
    TVertex*    GetFrom();
    TVertex*    GetTo();
    WeightType  GetWeight();
    void        SetFrom(TVertex* v);
    void        SetTo(TVertex* v);
    void        SetWeight(WeightType w);
```

CS101 Classes

CS101 classes are classes which implement classic computer science data structures. These include hash tables, linked lists, trees, graphs, etc. In general, you shouldn't use these classes directly - they are used as implementations for the collection classes. The collection classes isolate you from a particular implementation used for a collection (which will be a CS101 class). For example, a set could be implemented as a linked list, a c-array or a hash table depending on the operations performed, number of elements, etc. Eventually, the collection classes will be smart about making these choices for you automatically based on a specification from the user (the size hint is a start in that direction).



THashTable

The class `THashTable` is a subclass of `MCollectible` that can efficiently map an `MCollectible` to another `MCollectible`. Every hash table has a set of entries which associate a key to a value. Hash tables provide for fast lookup of a value when given a key even if there are a large number of entries in the table. Functions are provided for the usual things (insert, delete, etc.) as well as for controlling when rehashing will occur and the growth of the table when a rehash occurs. If you are using an `IsEqual` test function, be sure to override `Hash()` and `IsEqual()`. Iterators (see page 35) on `THashTables` return `TAssoc` objects. You can use the `GetKey()` and `GetValue()` call on the `TAssoc` to get what you want.

```
const long    kDefaultHashTableSize;
const long    kDefaultGrowthRate;
const long    kDefaultRehashThreshold;

class THashTable: public MCollectible {
 public:
   THashTable( MCollectibleCompareFn testFn = &MCollectible::IsSame,
               long tablesize=kDefaultHashTableSize,
               long growthrate=kDefaultGrowthRate,
               long threshold=kDefaultRehashThreshold);
   virtual                    ~THashTable();
```

```
  virtual long                  Count() const;
  virtual MCollectible*         Remove(const MCollectible& key);
  virtual MCollectible*         Delete(MCollectible* key);
  virtual MCollectible*         Member(const MCollectible& key) const;
  virtual void                  DeleteAll();
  virtual void                  DeleteAllKeys();
  virtual void                  DeleteAllValues();
  virtual void                  RemoveAll();
  virtual void                  Grow();
  virtual MCollectible*         Add(const MCollectible* key,
                                    MCollectible* value, boolean replace=TRUE);
  virtual MCollectible*         Retrieve(const MCollectible* key);
  virtual long                  GetGrowthRate() const;
  virtual long                  GetRehashThreshold() const;
  virtual void                  SetGrowthRate(long rate);
  virtual void                  SetRehashThreshold(long threshold);
  virtual MCollectibleHashFn    GetHashFunction(MCollectibleHashFn) const;
  virtual void                  SetHashFunction(MCollectibleHashFn);
}
```

`THashTable::THashTable(MCollectibleCompareFn testFn,`
`long tablesize, long growthrate, long threshold)`
Create a new hash table with an initial tablesize, growthrate and threshold. Return the new hashtable. TestFn is used by all methods when testing for keys which match. Threshold is a number from 0 to 100 which represents the maximum full percentage of the hash table.

`long THashTable::Count()`
Return the number of entries in the hash table. Initially and after a call to either `DeleteAll`, `RemoveAll`, `DeleteAllKeys` or `DeleteAllValues`, the count is zero.

`MCollectible* THashTable::Remove(const MCollectible& key)`
Remove any entry for the key in the hash table. Return the value removed if there was actually an entry to remove or `NIL` otherwise. It is important that you manage the storage for the value removed. If you are using an `IsEqual` comparison test, the key that is passed in might not be "the same" as the key in the hash table (only "equal"). This means that you will have a storage leak if the hash table had the only reference to the actual key and you remove the key, value pair using a surrogate key. In cases such as this, use the `Delete` method if you want the hash table to delete the actual key before returning the value. You could also call the `Member` method to retrieve the actual key in the dictionary before removing it.

`MCollectible* THashTable::Delete(MCollectible* key)`
Delete any entry for the `key` in the hash table and removes the value from the hash table. Return the value removed if there was actually an entry to remove or `NIL` otherwise. If the `key` used as a parameter is the same as the key in the hashtable, it is still deleted. Any operations on this deleted object will cause the usual storage management problems.

`MCollectible* THashTable::Member(const MCollectible& key) const`
Each object in this is compared to key using the function testFn. Return the object which was found which `IsEqual` or `IsSame` to the object passed as a parameter. Return NIL if no object was found.

`void THashTable::RemoveAll()`
Remove all of the entries in the hash table. Reset the count to be zero. If you don't have pointers to

all of the key, value pairs stored elsewhere in your program, you have a memory leak. You can use `DeleteAll`, `DeleteAllKeys` or `DeleteAllValues` if you would like the utility classes to destroy the objects in the hashtable.

`void THashTable::DeleteAll()`
Remove all of the entries in the hash table. Reset the count to be zero. Call the destructor on every key and every value in the hash table. If you have a key which also appears as a value or a value which appears more than once, you will be sorry you used this method because the utility classes will delete the same object more than once. This is not good.

`void THashTable::DeleteAllKeys()`
Remove all of the entries in the hash table. Reset the count to be zero. Call the destructor on every key in the hash table.

`void THashTable::DeleteAllValues()`
Remove all of the entries in the hash table. Reset the count to be zero. Call the destructor on every value in the hash table. If you have a value which appears more than once, you will be sorry you used this method because the utility classes will delete the same object more than once. This is not good.

`void THashTable::Grow()`
Force the hash table to grow by the rehashsize.

```
MCollectible* THashTable::Add(const MCollectible* key, MCollectible* value,
                    boolean replace=TRUE)
```
If replace=FALSE then only add key, value pair to the table if there is not an existing key,value pair. Otherwise, if replace=TRUE, add the key, value pair to the hash table. Either way, return the key that existed (if any) in the hash table before this call. Proper memory management may involve checking to see if the key returned is "the same" as the key passed in when replacing key, value pairs.

`long THashTable::GetGrowthRate() const`
Return the growth rate for the hash table.

`long THashTable::GetRehashThreshold() const`
Return the rehash threshold for the hash table. Threshold is a number from 0 - 100.

`void THashTable::SetGrowthRate(long rate)`
Set the growth rate for the hash table.

`void THashTable::SetRehashThreshold(long threshold)`
Set the rehash threshold for the hash table.

`MCollectibleHashFn THashTable::GetHashFunction() const`
Return the hash function being used by the hash table.

`void THashTable::SetHashFunction(MCollectibleHashFn)`[3]
Set which member function to call as a hash function. By default, this is set to `&MCollectible::Hash` (which is usually overridden in the objects you are adding to the hash table). You can use any hash function that you like as long as it has the type signature of `MCollectibleHashFn` (which is basically a method taking no parameters and returning a long).

3. D11 may not allow you to set the hash function after there are objects in the hash table. This bug will be fixed shortly by rehashing the table immediately after a `SetHashFunction` call.

Most of the time, you won't need to do this.

TLinkedList

A linked list object is useful for storing lists of `MCollectibles`. Linked lists are particularly useful when storage requirements are unpredictable and extensive manipulation of the structure (insertions, deletions) is required. Objects which are inserted into the linked list should override the `IsSame()` or `IsEqual()` method depending on what test function is passed into the constructor.

```
class TLinkedList: public MCollectible {
 public:
  TLinkedList(MCollectibleCompareFn testFn = &MCollectible::IsSame);
  virtual                        ~TLinkedList();
  virtual  long                  Count() const;
  virtual  MCollectible*         Remove(const MCollectible& obj, boolean
                                   removeAll=FALSE);
  virtual  MCollectible*         RemoveAfter(const MCollectible& obj);
  virtual  MCollectible*         RemoveBefore(const MCollectible& obj);
  virtual  void                  RemoveAll();
  virtual  MCollectible*         RemoveFirst();
  virtual  MCollectible*         RemoveLast();
  virtual  boolean               DeleteAll();
  virtual  boolean               AddAfter(const MCollectible& existingObj,
                                 MCollectible* obj);
  virtual  boolean               AddBefore(const MCollectible& existingObj,
                                 MCollectible* obj);
  virtual  void                  AddFirst(MCollectible* obj);
  virtual  void                  AddLast(MCollectible* obj);
  virtual  MCollectible*         After(const MCollectible& obj) const;
  virtual  MCollectible*         Before(const MCollectible& obj) const;
  virtual  MCollectible*         First() const;
  virtual  MCollectible*         Last() const;
  virtual  void                  Rotate(boolean firstBecomesLast=TRUE);

 protected:
           TLink*                GetFirst();
           TLink*                GetLast();
           void                  SetFirst(TLink*);
           void                  SetLast(TLink*);
  virtual  TLink*                MakeNewLink(TLink* n = NIL,
                                             MCollectible* v = NIL) const;
  virtual  TLink*                FirstLink();
  virtual  TLink*                LastLink();
  virtual  TLink*                Remove(TLink* l);
  virtual  TLink*                RemoveAfter(TLink* l);
  virtual  TLink*                RemoveBefore(TLink* l);
  virtual  boolean               AddAfter(TLink* l, MCollectible* obj);
  virtual  boolean               AddBefore(TLink* l, MCollectible* obj);
}
```

`TLinkedList::TLinkedList(MCollectibleCompareFn testFn);`
Create a new linked list and return it as the result. testFn is used by all methods when testing for

entries which match.

```
TLinkedList::~TLinkedList();
```
Destroy the linked list and all the links associated with it; however, the objects in the linked list are not freed. (The user is responsible for the objects - the system is responsible for the links).

```
long TLinkedList::Count();
```
Return the number of elements in the linked list.

```
MCollectible* LinkedList::Remove(const MCollectible& obj,
                                 boolean removeAll=FALSE);
```
Remove the first link which contains obj as the value of its link from the list. If removeAll=TRUE then remove all links which contain obj as its value. Return the removed object.

```
MCollectible* LinkedList::RemoveAfter(const MCollectible& obj);
```
Remove the link after the link containing obj. Return the removed object.

```
MCollectible* TLinkedList::RemoveBefore(const MCollectible& obj);
```
Remove the object before the link containing obj. Return the removed object.

```
void TLinkedList::RemoveAll();
```
Remove all of the objects in the list.

```
MCollectible* TLinkedList::RemoveFirst();
```
Remove the first object in the list. Return the removed object.

```
MCollectible* TLinkedList::RemoveLast();
```
Remove the last object in the list. Return the removed object.

```
void TLinkedList::DeleteAll();
```
Delete all of the links in the list. Free all of the objects in the list (that is, call the destructor on each object) as well as the links used to hold the objects.

```
oolean TLinkedList::AddAfter(const MCollectible& existingObj,
                             MCollectible* obj);
```
Add obj after existingObj in the list. If existingObj does not actually exist in the list then return false; otherwise return true.

```
boolean TLinkedList::AddBefore(const MCollectible& existingObj,
                               MCollectible* obj);
```
Add obj before existingObj in the list. If existingObj does not actually exist in the list then return false; otherwise return true.

```
void TLinkedList::AddFirst(MCollectible* obj);
```
Add obj to the front of the list in a newly created link.

```
void TLinkedList::AddLast(MCollectible* obj);
```
Add the obj to the end of the list

```
MCollectible* TLinkedList::After(const MCollectible& obj) const;
```
Return the object after the first occurrence of obj in the list. If there is no object after obj then return NIL.

`MCollectible* TLinkedList::Before(const MCollectible& obj) const;`
Return the object before the first occurrence of obj in the list. If there is no object before obj then return NIL.

`MCollectible* TLinkedList::First() const;`
Return the first object in the list. If there are no objects in the list, return NIL.

`MCollectible* TLinkedList::Last() const;`
Return the last object in the list. If there are no objects in the list, return NIL.

`void TLinkedList::Rotate(boolean firstBecomesLast=TRUE);`
If firstBecomesLast=TRUE, the first element in the list becomes the last element, the second becomes the first, the third becomes the second, and so on. If firstBecomesLast=FALSE, the last element becomes the first element, the first becomes the second, and so on.

`TLink* TLinkedList::MakeNewLink(TLink* n = NIL, MCollectible* v = NIL) const`
Whenever the `TLinkedList` class needs to make a new link, this virtual function is called. The default implementation calls `new TLink(n, v)`. Subclasses can override this method if they want their own kind of `TLink` created instead.

`TLink* TLinkedList::GetFirst();`
Return the first link in the list. If there are no links in the list, return NIL.

`TLink* TLinkedList::GetLast();`
Return the last link in the list. If there are no links in the list, return NIL.

`TLink* TLinkedList::SetFirst(TLink* aLink);`
Sets the first element of the linked list to be `aLink`.

`TLink* TLinkedList::SetLast(TLink* aLink);`
Sets the last element of the linked list to be `aLink`.

`TLink* TLinkedList::Remove(TLink* l);`
Remove the link l from the list. If it doesn't exist in the list then return NIL. Otherwise, if successful, return the deleted link. For this and all other delete operations on links, the user is responsible for freeing the deleted link. Users should be sure they know what they are doing before using functions which operate on links directly. The destructor function for linked lists will destroy all of the links. Therefore, if user functions cache a link reference, they may have a dangling pointer.

`TLink* LinkedList::RemoveAfter(TLink* l);`
Remove the link after l. If l is the last link in the list or l is not part of the list then return NIL; otherwise return the deleted link.

`TLink* TLinkedList::RemoveBefore(TLink* l);`
Remove the link before l. If l is the first link in the list or l is not part of the list then return NIL; otherwise return the deleted link.

`boolean TLinkedList::AddAfter(TLink* l, MCollectible* obj);`
Add obj after the link l. Create a new link to contain the obj. If l is not in the list then return false; otherwise return true.

`bboolean TLinkedList::AddBefore(TLink* l, MCollectible* obj);`
Add obj before the link l. Create a new link to contain obj. If l is not in the list then return false;

otherwise return true.

```
TLink* TLinkedList::FirstLink();
```
Return the first link in the list. If there are no links in the list, return NIL.

```
TLink* TLinkedList::LastLink();
```
Return the last link in the list. If there are no links in the list, return NIL.

TDoubleLinkedList

A `TDoubleLinkedList` object is useful for storing lists of `MCollectibles`. `TDoubleLinkedLists` are particularly useful when storage requirements are unpredictable and extensive manipulation of the structure (insertions, deletions) is required. Also, they are much more efficient (in time) than singly linked lists (`TLinkedList`), for two reasons. First, caching is done to remember which was the last accessed object in the list. Then, because the list is doubly linked, operations such as `Before` (as well as `After`, of course) are efficient.[4] Objects which are inserted into the linked list should override the `IsSame()` or `IsEqual()` method.

```
class TDoubleLinkedList {
 public:
   TDoubleLinkedList(MCollectibleCompareFn testFn = &MCollectible::IsSame);
   virtual ~TDoubleLinkedList();
   virtual  long             Count() const;
   virtual  MCollectible*    Remove(const MCollectible* obj,
                                    boolean removeAll=FALSE);
   virtual  MCollectible*    RemoveAfter(const MCollectible* obj);
   virtual  MCollectible*    RemoveBefore(const MCollectible* obj);
   virtual  void             RemoveAll();
   virtual  MCollectible*    RemoveFirst();
   virtual  MCollectible*    RemoveLast();
   virtual  boolean          DeleteAll();
   virtual  boolean          AddAfter(const MCollectible& existingObj,
                                     MCollectible* obj);
   virtual  boolean          AddBefore(const MCollectible& existingObj,
                                      MCollectible* obj);
   virtual  void             AddFirst(MCollectible* obj);
   virtual  void             AddLast(MCollectible* obj);
   virtual  MCollectible*    After(const MCollectible& obj) const;
   virtual  MCollectible*    Before(const MCollectible& obj) const;
   virtual  MCollectible*    First() const;
   virtual  MCollectible*    Last() const;
   virtual  void             Rotate(boolean firstBecomesLast=TRUE);

 protected:
   virtual  TDoubleLink*     MakeNewLink(TDoubleLink* previous,
                                        TDoubleLink* next,
                                       MCollectible* value = NIL) const;
 }
```

4. Unfortunately, this caching can cause some ambiguities when multiple instances of the same object are in the `TDoubleLinkedList`. At this point this is not viewed as a problem, only a feature that people need to be aware of.

All of the methods of `TDoubleLinkedList` behave like their counterparts in `TLinkedList`.

TTree

A `TTree` is an abstract superclass used as a baseclass for binary trees. TTrees provide some ordering on their members. Objects which are added to trees should be descended from `MOrderableCollectible`. Objects which are inserted into the TTree should override the `IsEqual()`, `IsLessThan()` and `IsGreaterThan()` methods.

```
class TTree: public MCollectible {
  public:
     TTree();
     virtual                          ~TTree();
     virtual long                     Count() const;
     virtual void                     Add(MOrderableCollectible* obj);
     virtual MOrderableCollectible*   Remove(const MOrderableCollectible& obj);
     virtual MOrderableCollectible*   First() const;
     virtual MOrderableCollectible*   Last() const;
     virtual void                     RemoveAll();
     virtual void                     DeleteAll();
     virtual MOrderableCollectible*   Member(const MOrderableCollectible& obj)
                                                const;
  }
```

`TTree::TTree();`
Create a new tree. Use `testFn` to determine where in the tree to perform insertion, searches, etc. (A test of `IsLessThan` means that "smaller" objects will end up to the left of the root and larger objects will end up to the right. Using a test of `IsGreaterThan` reverses this).

`void TTree::Add(MOrderableCollectible* obj)`
Add obj to the tree.

`MOrderableCollectible* TTree::Remove(const MOrderableCollectible& obj)`
Remove the obj in the tree that "is equal" to the passed in object. Return the object removed from the tree or NIL if no object was removed.

`void TTree::RemoveAll()`
Remove all the objects from the tree.

`void TTree::DeleteAll()`
Delete all the objects from the tree. Also, deallocate (i.e. call the destructor) on each object in the tree.

`MOrderableCollectible* TTree::Member(const MOrderableCollectible& obj) const`
Returns the actual object that is in the tree if the passed in obj "IsEqual" to an object in the tree. Returns NIL otherwise.

`MOrderableCollectible* TTree::First() const`
Returns the first object in the tree.

```
MOrderableCollectible* TTree::Last() const
```
Returns the last object in the tree.

TBinaryTree

A TBinaryTree is a subclass of tree. Each node in a binary tree can hold only one key object, a pointer to its left child and a pointer to its right child. Objects which are inserted into the TBinaryTree should override the IsEqual(), IsLessThan() and IsGreaterThan() methods. These are used internally by the TBinaryTree class.

```
class TBinaryTree: public TTree {
 public:
   TBinaryTree(MOrderableCompareFn fn=&MOrderableCollectible::IsLessThan);
   virtual ~TBinaryTree();
 }
```

```
TBinaryTree::TBinaryTree( MOrderableCompareFn
                        testFn=&MOrderableCollectible::IsLessThan);
```
Use testFn to determine where in the tree to perform insertion, searches, etc. (A test of IsLessThan means that "smaller" objects will end up to the left of the root and larger objects will end up to the right. Using a test of IsGreaterThan reverses this).

TRedBlackTree

A TRedBlackTree is an subclass of TBinaryTree which guarantees that the tree is always balanced. This removes any worst case abhorrent behavior for searching, inserting, and deleting on normal binary trees. Objects which are inserted into the TRedBlackTree should override the IsEqual(), IsLessThan() and IsGreaterThan() methods. These are used internally by the TBRedBlackTree class.

```
class TRedBlackTree: public TBinaryTree {
 public:
   TRedBlackTree(TOrderableCompareFn fn=&MOrderableCollectible::IsLessThan);
   virtual ~TRedBlackTree();
 }
```

```
TRedBlackTree::TRedBlackTree( TOrderableCompareFn
                              testFn=&MOrderableCollectible::IsLessThan);
```
Use testFn to determine where in the tree to perform insertion, searches, etc. (A test of IsLessThan means that "smaller" objects will end up to the left of the root and larger objects will end up to the right. Using a test of IsGreaterThan reverses this).

THeap

A THeap is a data structure which insures that the elements of the heap are always partially ordered and balanced. Because heas are only partially ordered, they can be more efficient than RedBlackTrees if the type of behavior that you want is to be able to add some objects to the heap and then remove the largest, then add some more, remove next largest, etc. Objects which are inserted into the THeap should override the IsEqual(), IsLessThan() and IsGreaterThan() methods.

```
const long kInitialHeapSize;

class Heap: public MCollectible {
 public:
   THeap(TOrderableCompareFn testFn=&MOrderableCollectible::IsLessThan,
          long heapSize = kInitialHeapSize);
   virtual                            ~THeap();
   virtual  MOrderableCollectible* Pop();
   virtual  MOrderableCollectible* Peek() const;
   virtual  long                   Count() const;
   virtual  void                   RemoveAll();
   virtual  void                   DeleteAll();
   virtual  void                   Add(MOrderableCollectible* obj);
   virtual  MOrderableCollectible* Remove(const MOrderableCollectible& obj);
   virtual  MOrderableCollectible* Member(const MOrderableCollectible& obj)
                                             const;
   virtual  MCollectibleCompareFn  GetEqualityComparisonFunction();
   virtual  void      SetEqualityComparisonFunction(MCollectibleCompareFn fn);
}
```

`THeap::THeap(TOrderableCompareFn testFn=&MOrderableCollectible::IsLessThan);`
Use testFn to determine whether larger objects are removed first or last. A test of `IsLessThan` means that larger objects are removed first and smaller objects are removed last. A test of `IsGreaterThan` reverses this.

`MOrderableCollectible* THeap::Pop()`
Remove the object at the top of the heap and return it.

`MOrderableCollectible* THeap::Peek() const`
Return the object at the top of the heap. This does not change what object is at the top of the heap.

`long THeap::Count() const`
Return a count of the number of objects in the heap.

`void THeap::Add(MOrderableCollectible* obj)`
Add obj to the heap.

`MOrderableCollectible* THeap::Remove(const MOrderableCollectible& obj)`
Remove obj from the heap. Return the actual object removed (which may not be the same as the object passed in only “is equal”) or NIL if no object was removed.

`MOrderableCollectible* THeap::Member(const MOrderableCollectible& obj) const`
Return true if obj is in the heap.

`void THeap::RemoveAll()`
Remove all the objects from the heap.

`void THeap::DeleteAll()`
Remove all the objects from the heap. Also, deallocate (i.e. call the destructor) on each object in the tree.

MCollectibleCompareFn THeap::GetEqualityComparisonFunction() const
Return the equality comparison function being used by the heap.

void THeap::SetEqualityComparisonFunction(MCollectibleCompareFn)
Set which member function to call as the equality comparison function when removing objects from the queue, checking to see whether a given object is a member, etc. This defaults to IsEqual. Most of the time you won't want to change this.

TGraphs

A TGraph provides an abstract superclass for all graphs. Objects which are inserted into the graph should override the Hash() method and the IsSame() method. These are used internally by the graph class.

```
const long  kExpectedNumberOfVertices = 20;
const long kExpectedAverageNumberOfEdgesPerVertex = 4;
class TGraph: public MCollectible {
  public:
   TGraph( const long vertices=kExpectedNumberOfVertices,
           const long edges=kExpectedAverageNumberOfEdgesPerVertex);
   virtual ~TGraph();
   virtual boolean   AddVertex(MCollectible* vertex, boolean replace = TRUE);
   virtual void      RemoveVertex(MCollectible* vertex);
   virtual void      AddEdge(MCollectible* from, MCollectible* to,
                             WeightType weight=1);
   virtual void      RemoveEdge(MCollectible* from, MCollectible* to,
                                WeightType weight=1);
   virtual TQueue*   ShortestPath(MCollectible* vertex1,
                             MCollectible* vertex2);
   virtual void      DepthFirstEach(TVertexActionFn fn, TQueue* start=NIL);
   virtual void      BreadthFirstEach(TVertexActionFn fn, TQueue* start=NIL);
   virtual boolean   IsComplete();
   }
```

TGraph(const long vertices, const long edges);
Create a new graph. While the graph which is returned could contain any number of vertices and edges, providing a guess as to the expected number of vertices and the expected average number of edges from each vertex could greatly improve the efficiency of graph operations.

~TGraph();
Delete the graph and all the vertices and edges associated with the graph.

boolean TGraph::AddVertex(MCollectible* vertex, boolean replace);
Add vertex to this. If vertex already exists in the graph and replace = TRUE then delete the old vertex and add the new vertex.

void TGraph::RemoveVertex(MCollectible* vertex);
Remove vertex from this.

void TGraph::AddEdge(MCollectible* from, MCollectible* to, WeightType weight);
Add the edge to the graph.

```
void TGraph::RemoveEdge(MCollectible* from, MCollectible* to, WeightType
weight);
```
Remove the edge from the graph.

```
TQueue* TGraph::ShortestPath(MCollectible* vertex1, MCollectible* vertex2);
```
Return the path in the graph connecting vertex1 and vertex2 with the property that the sum of the weights of the edges is minimized over all such paths. Each RemoveFirst operation on the queue will remove edges starting at vertex1 and moving to vertex2. The user is responsible for freeing this queue of vertices and edges when it is no longer needed.

```
void TGraph::DepthFirstEach(TVertexActionFn fn, TQueue* start=NIL);
```
Iterate over all of the vertices in the graph reachable from the start collection in a depth-first fashion. Apply fn to each vertex in the graph in this order. If start=NIL then an appropriate starting set of vertices will be chosen.

```
void TGraph::BreadthFirstEach(TVertexActionFn fn, TQueue* start);
```
Iterate over all of the vertices in the graph reachable from the start collection in a breadth-first fashion. Apply fn to each vertex in the graph in this order. If start=NIL then an appropriate starting set of vertices will be chosen.

```
boolean TGraph::IsComplete();
```
Return true if there is an edge from every vertex to every other vertex in the graph.

TUndirectedGraph

A `TUndirectedGraph` provides an implementation for an undirected graph. Objects which are inserted into the graph should override the `Hash()` method and the `IsSame()` method. These are used internally by the `graph` class.

```
class TUndirectedGraph: public TGraph {
  public:
    TUndirectedGraph(const long vertices=kExpectedNumberOfVertices,
                     const long edges=kExpectedAverageNumberOfEdgesPerVertex);
    virtual ~TUndirectedGraph();
    virtual TSet*     MinimumSpanningTree();
    virtual boolean   IsConnected();
    virtual TSet*     BiconnectedComponents();
    virtual TDeque*   ConnectedComponents();
    virtual TSet*     ArticulationPoints();
    virtual boolean   IsBiconnected();
  }
```

```
TUndirectedGraph::TUndirectedGraph(const long vertices, const long edges);
```
Create a new graph. While the graph which is returned could contain any number of vertices and edges, providing a guess as to the expected number of vertices and the expected average number of edges from each vertex could greatly improve the efficiency of graph operations.

```
~TUndirectedGraph::TUndirectedGraph();
```
Delete the graph and all the vertices and edges associated with the graph. All edges and vertices are freed.

```
TSet* TUndirectedGraph::MinimumSpanningTree();
```
Return a set of edges which represents a minimum spanning tree of a weighted graph. A minimum spanning tree is a collection of edges that connects all the vertices such that the sum of the weights of the edges is at least as small as the sum of the weights of any other collection of edges that connects all the vertices. (Real life: I want to wire a group of cities so that each city can reach each other city and I want to minimize the amount of wire to use.) The user is responsible for freeing this set of vertices and edges when it is no longer needed.

```
boolean TUndirectedGraph::IsConnected();
```
Return true if this is connected. A graph is connected if there is a path from every vertex in the graph to every other vertex in the graph.

```
TSet* TUndirectedGraph::BiconnectedComponents();
```
Return a collection of biconnected components of this . Each object in the set is itself a graph. Biconnected components of a graph are sets of vertices mutually accessible via two distinct points. The user is responsible for freeing the returned set of vertices when it is no longer needed.

```
TDeque* TUndirectedGraph::ConnectedComponents();
```
Return a collection of connected components of this. Each object in the TDeque is itself a TDeque which represents a connected component of the graph. The user is responsible for freeing this TDeque of vertices and edges when it is no longer needed.

```
TSet* TUndirectedGraph::ArticulationPoints();
```
Return a collection of articulation points of this. Each object in the set is a vertex which is an articulation point (sometimes called a cutpoint). A vertex v is an articulation point for a graph if there are distinct vertices w and x (distinct from v) such that v is in every path from w to x. (A biconnected graph is one which has no articulation points - see below). The user is responsible for freeing the returned set of vertices when they are no longer needed.

```
boolean TUndirectedGraph::IsBiconnected();
```
Return true if this is biconnected. A graph is biconnected if and only if there are at least two different paths connecting each pair of vertices. Thus even if one vertex and all the edges touching it are removed, the graph is still connected. (Real life: We often want a network to be biconnected so that a failure along one point does not leave the whole system down)

DirectedGraphs

A `TDirectedGraph` provides an implementation for a directed graph. Objects which are inserted into the graph should override the `Hash()` method and the `IsSame()` method. These are used internally by the `graph` class.

```
class TDirectedGraph: public TGraph {
  public:
    TDirectedGraph( const long vertices=kExpectedNumberOfVertices,
                    const long edges=kExpectedAverageNumberOfEdgesPerVertex);
    virtual ~TDirectedGraph();
    virtual TQueue*    TopologicalSort(TQueue* start=NIL);
    virtual TSet*      StronglyConnectedComponents();
    virtual boolean    IsStronglyConnected();
    virtual boolean    IsWeaklyConnected();
    virtual boolean    IsAcyclic();
  }
```

```
TDirectedGraph::TDirectedGraph(const long vertices, const long edges);
```
Create a new directed graph (digraph). While the graph which is returned could contain any number of vertices and edges, providing a guess as to the expected number of vertices and the expected average number of edges from each vertex could greatly improve the efficiency of graph operations.

```
~TDirectedGraph::TDirectedGraph();
```
Delete the graph and all the vertices and edges associated with the graph. All edges and vertices are freed.

```
TQueue* TDirectedGraph::TopologicalSort(const TQueue* start=NIL);
```
Return an ordering on this such that no vertex in the ordering is before any vertex that points to it. Each object in the queue is a vertex. Use the start queue as the vertices to begin the topological sort. If no start queue is provided then some appropriate set of starting nodes will be chosen. (The appropriate set of nodes will consist of all nodes of in-degree zero) Note that there is no possible way to perform a topological sort on a graph which is not a directed acyclic graph (DAG). In this case, NIL would be returned. The user is responsible for freeing this queue of vertices when it is no longer needed.

```
TSet* TDirectedGraph::StronglyConnectedComponents();
```
Return a collection of strongly connected components of this. Each strongly connected component in the set is itself a DirectedGraph. (Strongly connected components of a graph are subgraphs in which there is a path from vertex v to vertex w for all vertices.). The user is responsible for freeing this set of graphs when it is no longer needed. Note: there may be a need for a strongly connected components in topological order function.

```
boolean TDirectedGraph::IsStronglyConnected();
```
A graph is strongly connected if for every pair of vertices v and w, there is a path from v to w (edges may only be traversed from tail to head).

```
boolean TDirectedGraph::IsWeaklyConnected();
```
A digraph is weakly connected if for every pair of vertices v and w, there is a path from v to w (edges may be traversed from tail to head or from head to tail - i.e., treat digraph as if it was undirected for purposes of this test).

```
boolean TDirectedGraph::IsAcyclic();
```
Return true if this graph has no cycles.

Random Numbers

TRandomNumberGenerator generates a sequence of pseudo random numbers given an initial seed. If no initial seed is specified, the system time is used as a seed. The range of random number values is [0, 2^31-1>.

```
class TRandomNumberGenerator {
 public:
   TRandomNumberGenerator();
   TRandomNumberGenerator(long initialSeed);
   ~TRandomNumberGenerator();
   long  Next();
   void  Reset();
   long  First();
```

```
  protected:
    long   GetSeed();
    void   SetSeed(long newInitialSeed);
};
```

`TRandomNumberGenerator::TRandomNumberGenerator()`
Construct a new random number generator. Use the system time as a seed value.

`TRandomNumberGenerator::TRandomNumberGenerator(long initialSeed)`
Construct a new random number generator using `initialSeed` as the seed.

`TRandomNumberGenerator::~TRandomNumberGenerator()`
Destroy the random number generator.

`long TRandomNumberGenerator::Next()`
Return the next random number in the sequence.

`long TRandomNumberGenerator::First()`
Equivalent to `Reset(); Next();`.

`void TRandomNumberGenerator::Reset()`
Reset the random number generator. This can be used to replay a sequence of random numbers.

`long TRandomNumberGenerator::GetSeed()`
Get the initial seed used by the random number generator.

`void TRandomNumberGenerator::SetSeed(long newInitialSeed)`
Set the seed used by the random number generator.

Advanced Topics

Iterators

Earlier versions of the utility classes included an "Each" mechanism for iterating over the objects in a class. Unfortunately, there are a number of problems with this mechanism (difficult to pass back information and no closures in C++) that facilitate the need for a more generic mechanism.

All of the classes described in the document have iterator classes defined for them. An iterator for a particular object will iterate over all of the objects in a class. For example, the TLinkedListIterator will iterate over each element in the TLinkedList class. Each call to the iterator will return the next element in the class. For example:

```
TLinkedList alist = new TLinkedList();
// Some linked list adds, etc.
TLinkedListIterator* iterator = new TLinkedListIterator(alist);
MCollectible* foo = iterator->First();
while (foo != NIL)
{
  // do something to foo
  foo = iterator->Next();
}
```

The virtual function `Next()` is used to "get the next object from this class." Iterators are used internally to implement functions such as Each, Member, Some and Every. For example, the definition of `Some` (a method which returns true if some element of a collection passes a specific test) could be:

```
{
  TIterator* i = Iterator();
  MCollectible* e;
  boolean done = false;

  if (i != NIL)
  {
    e = i->First();
    while ((e != NIL) && (done == false))
    {
      done = (e->*fn)(some arguments....);
      e = i->Next();
    }
    delete i;
  }
  return done;
}
```

Objects in the class will be returned with order preserved if the class contains objects which are fully ordered. For example, linked lists, deques, queues, stacks, binary trees, etc. will return objects "in order." Hashtables, sets, bags, dictionaries, heaps, priority queues etc. will return objects in some random (at least to the user) order.

Operations on the collection itself will invalidate all outstanding iterators on the collection until the iterator resyncs with the collection. This occurs in the calls `First` and `Last`. Starting with d11

Pink, you can remove an element from the collection using a method of the iterator (Remove). This automatically resyncs the iterator to the collection; however, all other iterators are invalidated just as if any operation was done to the collection they are tied to.

```
class TIterator {
 public:
   TIterator();
   virtual ~TIterator();
   virtual MCollectible*   First();
   virtual MCollectible*   Next();
   virtual MCollectible*   Remove();
 }
```

`TIterator::TIterator()`
Create a new iterator.

`TIterator::~TIterator()`
Delete the iterator.

`MCollectible* TIterator::First()`
Reset the iterator and return the first element of the collection. This resyncs the iterator to the collection if other operations on the collection caused the iterator to be invalidated.

`MCollectible* TIterator::Remove()`
Remove the current object (the one just returned by first or next) from the collection. This is the only way to remove an object from a collection while iterating. All other iterators are invalidated just as if any change to the collection had occurred. If the collection has changed (other than through the use of the `Remove` method of this iterator) since the last time `First` was called, this method will `FAIL`.

`MCollectible* TIterator::Next()`
Retrieve the next object in the collection and return it. The order that objects are retrieved is in an order that reflects the "ordered-ness" of the collection (or the lack of ordering on the collection elements). If the collection has changed (other than through the use of the `Remove` method of this iterator) since the last time `First` was called, this method will `FAIL`.

Subclasses of `TSequence` can also use a `TSequenceIterator` for iterating through the objects in a collection in "backwards" order rather than the usual order.

```
class TSequenceIterator : public TIterator {
 public:
   TSequenceIterator();
   virtual ~TSequenceIterator();
   virtual MCollectible*   Last();
   virtual MCollectible*   Previous();
 }
```

`TSequenceIterator::TSequenceIterator()`
Create a new iterator.

`TSequenceIterator::~TSequenceIterator()`
Delete the iterator.

`MCollectible* TSequenceIterator::Last()`
Return the last object in the collection. This resyncs the iterator to the collection if other operations on the collection caused the iterator to be invalidated.

`MCollectible* TSequenceIterator::Previous()`
Retrieve the previous object in the collection and return it. The order that objects are retrieved is in an order that reflects the "ordered-ness" of the collection (or the lack of ordering on the collection elements). If the collection has changed (other than through the use of the `Remove` method of this iterator) since the last time `First` or `Last` was called, this method will `FAIL`.

Garbage Collection

The role of automatic storage management in the C++ world is a very controversial issue. In a recent paper, "Possible Directions for C++," Stroustrup states that garbage collectors and C++ are probably not a good fit. Stroustrup argues, "It seems unlikely that simply applying the garbage collection techniques perfected for languages designed with garbage collection in mind to C++ would make C++ a good garbage collected language. It might be done, but I suspect that since the fundamental design choices for C++ were made assuming the absence of a garbage collector, a C++ garbage collector would be less efficient than, say, a good Lisp or CLU garbage collector. Worse, I suspect that a garbage collected C++ would fail to deliver the low level performance people have come to expect from C++."

Stroustrup adds that a more appropriate course of action would be to continue the C++ approach to giving the programmer lots of choices for allocating objects with something that could be seen as a "garbage collected class." This would allow the convenience (said in an appropriate Church Lady tone) of automatic storage with the unrestricted lifetimes of objects on the free store. He then alludes to some work being done by Jonathan Shopiro on his "counted pointer" classes. A counted pointer is an object that acts like a pointer. Counted pointer objects are counted and when the last counted pointer to an object is destroyed, the object itself is destroyed (this is similar to a reference count garbage collector).

Earlier versions of the utility classes allowed the use of counted pointer objects with the utility classes. This is now viewed as an experiment that failed. This is not to say that garbage collection is unimportant; however, counted pointers are not the answer. True garbage collection would increase the usability of these classes substantially.

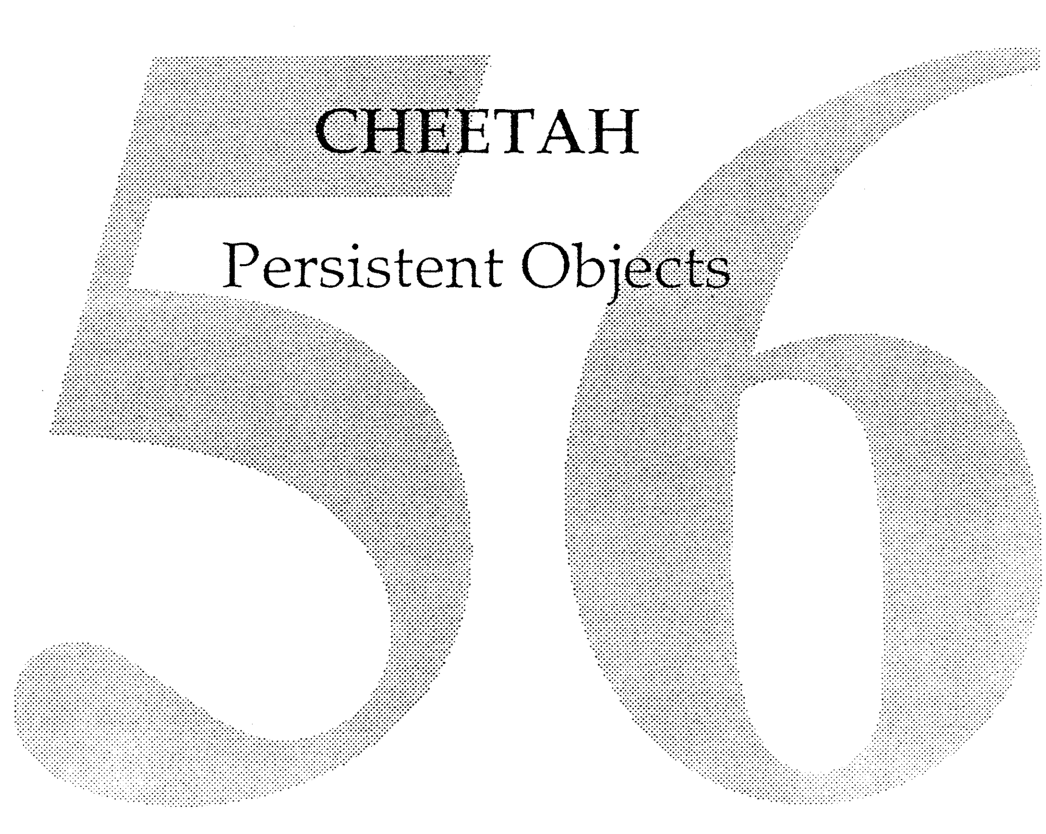
As it stands now, users of these classes should take great care with respect to memory management. In general, the utility classes manage their own memory and never allocate memory that they expect the user to manage.[5] Likewise, objects created by the user and put into collections should be managed by that user. A common error that many people encounter is the following:

```
TSet* aSet = new TSet();
TSurrogateTask* aTask = new TSurrogateTask();
aSet->Add(aTask);


...much later...
```

5. The major exception to this rule is the member function, `Iterator()`, which creates a new iterator on the heap that it expects the user to manage. If you don't need the polymorphism (that is, you know the type of the collection as something other than `TCollection`), you can create an iterator directly as in `TDequeIterator anIterator(&aDeque)`.

```
TSurrogateTask* bTask = new TSurrogateTask();
// bTask is equal to aTask but not the same object

// The wrong way unless you know bTask is pointer eq to aTask
aSet->Remove(bTask);
delete bTask;

// The right way in general
TSurrogateTask* someTask = (TSurrogateTask*) aSet->Remove(bTask);
if (someTask != bTask)
  delete someTask;
delete bTask;
```

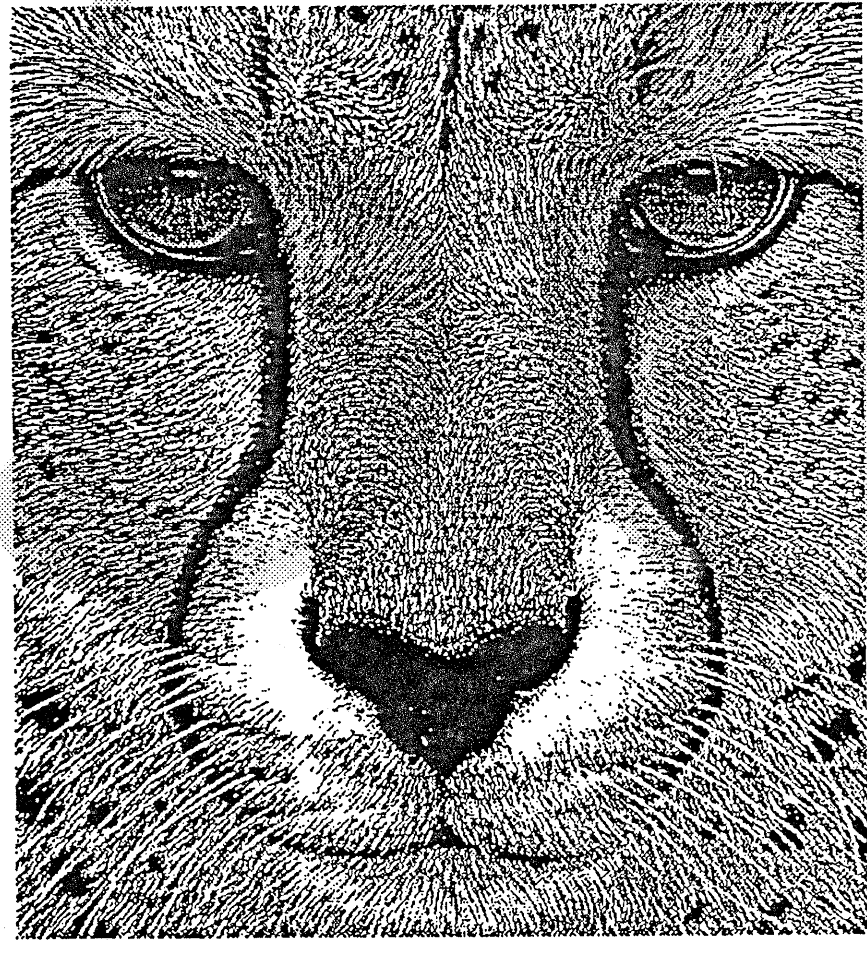
Dictionaries are slightly more complicated to deal with because you always receive the old value back from the call to `Remove`. This means that the pointer to the key is dropped on the floor by the utility classes. Use the `Delete` call to remove the `key, value` pair, delete the key and return the value to the caller.


CHEETAH
Persistent Objects

56

chee•tah *n.* **1.** A long-legged, swift running wild cat, *Acinonyx jubatus*, of Africa and southwestern Asia, that has black spotted, tawny fur and nonretractile claws and is sometimes trained to pursue game. **2.** Cheap persistent objects for Pink.

Arn Schaeffer
x48117



56

Introduction

cheetah is a set of classes and protocol for use with C++ in the Pink world for saving and restoring objects to and from a stream. Objects which descend from the appropriate classes and adhere to the proper protocol can be "flattened" to a stream (memory, a disk file, the network, etc.) and "expanded" on the other end.

Objects with references to other objects can be flattened and restored easily. Multiple references to the same object are restored properly. Circular references are also handled in the design of the system. Sets of objects can be flattened and restored together and repeated references are handled efficiently. cheetah does not address where objects are stored, how they are found, indexing into a database of objects, or garbage collection of persistent objects.

Architecture Overview

Converting objects to a "flattened" form is a desirable feature for an object oriented programming system. It allows objects to have a persistent state which can span across processes, sessions and CPUs. cheetah provides a set of classes and protocol which can be used by classes which need this capability. Instances of objects from classes derived from the class `MCollectible` can be flattened to a persistent state using a global function, `FlattenPointer`. This function grovels over the structure of the object and all objects referenced by this object. This process results in a linearized form of the object which can be stored on disk, sent to another CPU, etc. The reverse of this process is to use the global function `Resurrect which` takes a flattened object and creates an `MCollectible` object as a result.

The architecture of cheetah is set up to allow efficient representations of flattened objects which are ephemeral (in the case of a flattened object which is packed in an Opus/2 message where space is a concern) as well as efficient representations of flattened objects which are persistent (those which are going to disk to be restored at a much later time). The flattening process considers the lifetime of this object when generating a flattened form for this object (or set of objects). Also, the system allows class designers to provide hints to the system concerning the nature of the object structure. For example, if the class designer knows that the structure of an object is a tree (that is, if one were to walk the structure of the object and all the objects that the object pointed to, there would be no repeated references), the class designer could tell the system this.

Recent additions to cheetah allow for the management of persistent pools of objects which can be randomly accessed. These persistent pools provide much of what a persistent object system provides.

Details

TStream

The `TStream` class provides an abstract protocol for reading and writing data structures. The stream can be a section of memory, an Opus Message, a disk file or anything else that allows binary representations of objects to be written to it. `TStream` is an abstract superclass. Derived classes of `TStream` should implement the protocol of `TStream`. All of the methods in `TStream` signal exceptions when bad things happen (for example, end of file is reached). All of the read/write methods use a buffered approach for reading and writing. Virtual functions are called when the buffer is full or empty at which time your stream can do whatever processing it needs. If no buffering is desired, the size of the buffer can be set to zero and your virtual functions will be called at every read or write. This design allows the code for reading and writing to be inline and efficient (no function calls to write or read from a stream except at overflow or underflow) while also allowing streams to have some virtual behavior.

```
typedef size_t StreamPosition;

class TStream : public MCollectible {
 public:
   virtual  ~TStream();

   // Non virtual reads and writes for all the primitive types:
   // char, short, long (signed and unsigned), float, extended, double, etc.

   virtual  void              Reset();
   virtual  StreamPosition    Position();
   virtual  void              Seek(StreamPosition position);
   virtual  void              SeekRelative(StreamPosition amount);
   virtual  StreamPosition    GetLogicalEndOfStream();
   virtual  StreamPosition    GetPhysicalEndOfStream();
   virtual  void              SetContext(TContext*);
   virtual  TContext*         GetContext();
   virtual  void              SetDeepFreeze(Boolean);
   virtual  Boolean           GetDeepFreeze();
   virtual  void              SetDeferredWriteList(TDeque*);
   virtual  TDeque*           GetDeferredWriteList() const;
   virtual  void              SetForceFlattenEternalObjects(Boolean);
   virtual  Boolean           GetForceFlattenEternalObjects() const;

 protected:
   TStream(void* bufferStart, StreamPosition howmuch);
   virtual  void              BufferFull(const void*, StreamPosition);
   virtual  void              BufferEmpty(void*, StreamPosition);
            void*             GetBufferStart() const;
            void              SetBufferStart(void*);
            void              IncrementBufferStart(StreamPosition);
            StreamPosition    GetBufferLength() const;
            void              SetBufferLength(StreamPosition);
            void              DecrementBufferLength(StreamPosition);
            void              SetBufferWasModified(Boolean modified=TRUE);
            Boolean           GetBufferWasModified() const;
};
```

Overloaded operators for reading (you don't need to override these - in fact, you can't):

```
TStream& operator<<=(char* c, TStream& s);
TStream& operator<<=(long& c, TStream& s);
TStream& operator<<=(short& c, TStream& s);
TStream& operator<<=(char& c, TStream& s);
TStream& operator<<=(Boolean& c, TStream& s);
...
```

Overloaded operators for writing (you don't need to override these - in fact, you can't):

```
TStream& operator>>=(const char* c, TStream& s);
TStream& operator>>=(const long& c, TStream& s);
TStream& operator>>=(const short& c, TStream& s);
TStream& operator>>=(const char& c, TStream& s);
TStream& operator>>=(const Boolean& c, TStream& s);
...
```

TStream::TStream(void* bufferStart, StreamPosition howmuch)
You must pass the start of a buffer to use as well as the size of the buffer when creating stream objects. This buffer will be used for reading and writing. Subclasses will not necessarily export this information to their clients. To specify no buffering, passing NIL, 0 will work.

void TStream::Reset()
Reset the stream. The current position is set to zero.

StreamPosition TStream::Position()
Return the current position of the stream.

void TStream::Seek(StreamPosition position)
Seek to the specified position. The next read or write will take place from there.

void TStream::SeekRelative(StreamPosition position)
Seek (relative to where you are) by the specified amount. The next read or write will take place from there.

StreamPosition TStream::GetLogicalEndOfStream()
Return the logical end of the stream.

StreamPosition TStream::GetPhysicalEndOfStream()
Return the physical end of the stream.

void TStream::BufferFull(const void*, StreamPosition count)
This routine is called by cheetah when the specified buffer that you supplied is full. Appropriate action should be taken on this buffer (flush it to disk if it is a disk file, etc.) Furthermore, the request which could not be processed because the buffer would overflow is passed into this routine. This request (location, count) should be processed before returning (perhaps it is put into a new buffer). Finally, the buffer length and current buffer start should be set appropriately.

void TStream::BufferEmpty(void*, StreamPosition count)
This routine is called by cheetah when the specified buffer that you supplied does not contain enough data to process a request. Appropriate action should be taken on the buffer (flush to disk, throw away, etc.). Furthermore, the request which couldn't be processed should be serviced. Finally, the buffer length and current buffer start should be set appropriately upon exit.

void TStream::SetContext(TContext*)
TContext* TStream::GetContext()
Set/Get the context used in reading or writing objects to the stream. It is only necessary to set the context when writing a set of objects which has multiple references to the same objects.

void TStream::SetDeepFreeze(Boolean)
Boolean TStream::GetDeepFreeze()
Set/Get the "deepFreeze" used in reading or writing objects to the stream. If GetDeepFreeze returns FALSE, tokens are used to represent the class name. If GetDeepFreeze returns TRUE, strings are used. When flattening objects which will persist across sessions or machines, GetDeepFreeze should return TRUE.

void TStream::SetDeferredWriteList(TDeque*)
TDeque* TStream::GetDeferredWriteList() const
This list is used internally by the cheetah system.

```
void     TStream::SetForceFlattenEternalObjects(Boolean)
Boolean  TStream::GetForceFlattenEternalObjects() const
```
Set/Get the flag that determines whether "eternal" objects will be treated as if they were ordinary objects when writing to the stream. If the flag is true, "eternal" object references are flattened on the stream much like an ordinary object. If the flag is false, "eternal" object references are noted in the stream and the "eternal" object is written to where it belongs. More on this in the section on eternal objects.

```
void* TStream::GetBufferStart() const
void TStream::SetBufferStart(void*)
void TStream::IncrementBufferStart(StreamPosition)
StreamPosition TStream::GetBufferLength() const
void TStream::SetBufferLength(StreamPosition)
void TStream::DecrementBufferLength(StreamPosition)
void TStream::SetBufferWasModified(Boolean modified=TRUE)
Boolean TStream::GetBufferWasModified() const
```
These routines perform operations on the current buffer start pointer and buffer length pointer that are used to determine where the next streamed data is written. These setters/getters should be used in the BufferFull/BufferEmpty routine to reset the buffer pointer/length before exiting.

Global Functions

There are two functions in the system which can be used to flatten and expand objects.

```
void FlattenPointer(MCollectible* objToBiteBigOne, TStream* towhere);
```
To flatten an object to a stream, you use the `FlattenPointer` command. The stream to flatten the object to is passed in as well as the object to be flattened. If you wish to flatten multiple objects to a stream, set the context of the stream before making this call (see the section on streams). The `TContext` argument is used in the case where multiple objects will be flattened and packed together on the same stream. The `TContext` is basically a dynamic dictionary which is built during the flattening process to assign references to repeated object instances in the set of objects which is to be saved. If only a single object is to be saved, the `TContext` can be generated automatically by the system. If you wish to save multiple objects (which might point to overlapping objects which also need to be saved), you need to provide a `TContext`. For example:

```
// FlattenPointer a single object.
TPersistentClassA* a = new TPersistentClassA('a', 5);
TMemoryStream pneuma1(new char[100], 100);
FlattenPointer(a, &pneuma1);

// FlattenPointer two objects with shared parts
TPersistentClassD* d = new TPersistentClassD("me", "cguy", "bguy", a);
TMemoryStream pneuma2(new char[100], 100);
TContext tim;
pneuma2->SetContext(&tim);
FlattenPointer(d, &pneuma2);
FlattenPointer(a, &pneuma2);
pneuma2->SetContext(NIL);
```

Streams also provide information about whether they are ephemeral (in the case of a memory stream) or more persistent (as in a disk file). A `deepFreeze` attribute of the stream is set to true if the flattening should store the object in its most general form; that is, a form which can be resurrected on another CPU or saved to disk and resurrected. Objects which are simply sent to another team (for example, in an Opus/2 message) can use a more compact representation. See the examples section for code which saves single objects and multiple objects. Note that in any event, the original object is unchanged by the

process of flattening. The stream has the flattened version of the object at the time of the snapshot.

```
MCollectible* Resurrect(TStream* fromwhere);
```

The `Resurrect` function will take the flattened form of an object and create an `MCollectible` object from it. The stream which is passed in contains the flattened form of the object. The context which is passed in is used in the same manner as above. For example, to resurrect the two objects created in the example above:

```
// Resurrect a single object
TPersistentClassA* A;
pneuma1->Reset();
A = (TPersistentClassA*) Resurrect(pneuma1);

// Resurrect multiple objects from the same stream
TPersistentClassD* D;
pneuma2->Reset();
TContext context;
pneuma2->SetContext(&context);
D = (TPersistentClassD*) Resurrect(pneuma2);
A = (TPersistentClassA*) Resurrect(pneuma2);
pneuma2->SetContext(NIL);
```

It is of the utmost importance to note the order of resurrection must follow the order of flattening exactly in the case of saving and restoring sets of objects.

MCollectible

`MCollectible` should be mixed into a class to enable it to be flattened or resurrected. `MCollectible` provides a general mechanism for reading and writing objects to a stream.

It is only necessary to be a subclass of `MCollectible` if you wish to FlattenPointer or resurrect objects of that class directly. Classes with base classes or member objects which are not derived from `MCollectible` can still be flattened and resurrected as part of a derived class if these classes supply an `operator>>=` and `operator<<=` function (these operators don't need to be virtual if you aren't descending from `MCollectible`). Objects which are references (by a pointer) from an `MCollectible` object, must, of course, be `MCollectible` objects (because flattening and resurrecting these objects requires calls to `FlattenPointer` and `Resurrect`).

```
typedef     TokenID     ClassName;

class MCollectible[1] {
 public:
  MCollectible();
  virtual  ~MCollectible();
  virtual  TStream&    operator>>=(TStream& towhere);
  virtual  TStream&    operator<<=(TStream& towhere);
  virtual  StreamPosition    Size(TContext* tim = NIL, Boolean deepFreeze =
FALSE);
  virtual  ClassName   GetClassNameAsToken();
  virtual  char*       GetClassNameAsString();
```

1. This is the part of MCollectible concerned with flattening objects to a stream and resurrecting them later. For more information about MCollectible, see the Utility Classes document.

```
    virtual  Boolean     StructureDoesNotHaveRepeatedReferences(Boolean
                                     deepFreeze = FALSE);

    // These methods will be discussed later with MEternal.
    virtual  ObjectID               GetObjectID() const;
    virtual  void                   SetObjectID(ObjectID);
    virtual  TPersistentContext*    GetPersistentContext() const;
    virtual  void                   SetPersistentContext(TPersistentContext*);
    virtual  Boolean                GetDirty() const;
    virtual  void                   SetDirty(Boolean dirty = TRUE);
};
```

Note: When subclassing from MCollectible, include the line:
`MCollectibleDeclarationsMacro(myClassName);`
in the declaration of your class and the line:
`MCollectibleDefinitionsMacro(myClassName);`
in your .c file.

`TStream& MCollectible::operator<<=(TStream* fromwhere)`
This is a workhorse routine. Every `MCollectible` object must have a method like this one. This expand method is called by the `Resurrect` procedure as well as directly by the user to restore an object from its flattened form. For example, if class C is descended from classes A and B, has a member object d of class D, has a member long, a member char, and a pointer to an E object (e), its `Expand` routine would look like:

```
TStream& C::operator<<=(TStream* fromwhere)
{
    A::operator<<=(fromwhere);
    B::operator<<=(fromwhere);
    d.operator<<=(fromwhere);
    fLong <<= fromwhere;
    fChar <<= fromwhere;
    e = (E*) Resurrect(fromwhere);
    return fromwhere;
}
```

`TStream& MCollectible::operator>>=(TStream* towhere)`
This is the other workhorse routine. This routine must flatten the baseclasses of the object by explicitly calling the flatten routine of the baseclasses. It must then flatten the member objects by explicitly calling the flatten routine of the member objects. Finally, it must flatten its members which are simple data types and references to other objects. Simple data types are flattened by writing the datatype to the stream (using the `stream::Write()` member function). References to other objects are flattened by recursively calling the `FlattenPointer` routine with the parameters which were passed in. For example, if class C is descended from classes A and B, has a member object d of class D, has a member long, a member char, and a pointer to an E object (e), its `Flatten` routine would look like:

```
TStream& C::operator>>=(TStream* towhere)
{
    A::operator>>=(towhere);
    B::operator>>=(towhere);
    d.operator>>=(towhere);
    fLong >>= towhere;
    fChar >>= towhere;
    FlattenPointer(e, towhere);
    return towhere;
}
```

```
StreamPosition MCollectible::Size(TContext* tim = NIL, Boolean deepFreeze =
FALSE)
```
This method returns the size that this object will be when it is flattened in the passed in context. This routine should probably not be overridden except by the most stalwart subclasses. This is a very expensive method to call because cheetah needs to flatten the object in order to determine its size. If you are going to be flattening the object anyway, a better way of determining the size is to reserve space in the stream for the size of the object, flatten the object, note the position, seek back to where the size goes, compute the size (you know the final position and the initial position), and stream it.

```
const char* MCollectible::GetClassNameAsString()
```
Return the name of the class as a string.

```
ClassName MCollectible::GetClassNameAsToken()
```
The default version of this routine calls the GetClassNameAsString command, then asks the token manager for the token that matches this class. Subclasses can override this method to cache the token. This routine actually returns the tokenID of the token. This will be changed sometime after d11.

```
Boolean MCollectible::StructureDoesNotHaveRepeatedReferences
                                   (Boolean deepFreeze = FALSE)
```
This routine returns true if the structure does not have any repeated references to other objects (or a circular reference to itself). If true, there are some optimizations that can be done during flattening. If you don't know whether this is true, returning false is safer (which, incidentally, is the default).

TContext

`TContext` is a class which is used with the global functions, `FlattenPointer` and `Resurrect` to provide a mechanism for restoring sets of objects. The developer should never need to override any of these routines or ever create anything other than a default `TContext` object (that is, the one constructed by passing zero arguments to the constructor).

```
typedef     long     LocalObjectNumber;

class TContext {
 private:
   TDictionary*    fOtherObjects;
   Boolean         fMultipleObjectContext;
 public:
   TContext(Boolean multipleObjectContext = TRUE);
   virtual  ~TContext();
   virtual  long              Count();
   virtual  MCollectible*     Find(LocalObjectNumber);
   virtual  LocalObjectNumber Add(MCollectible*, Boolean& newentry);
   virtual  MCollectible*     Replace(MCollectible* newobject,
                                       LocalObjectNumber fred);
   virtual  Boolean           GetMultipleObjectContext();
};
```

Eternal Objects

The data management tools in the toolbox provide methods for the rapid retrieval of an arbitrary object given an arbitrary key. cheetah specifies the external data format for an object. The integration of the data management tools with cheetah provide for a simple persistent object system in which objects can be individualy addressed and brought into memory. A `TPersistentContext` is a pool of objects in which any object can be found and mapped into memory given its object id. Objects which live in a `TPersistentContext` have `MEternal` as a mixin. When `FlattenPointer` is called on an "eternal" object, a reference to the eternal object is put in the stream rather than the flattened version of the object itself. The eternal object (if dirty) is flushed to the `TPersistentContext` at this time. When `Resurrect` is called on a stream which contains a reference to an eternal object, the object is returned if already in memory or loaded from the `TPersistentContext` if not.

Designing a class of objects which mixin `MEternal` involves a number of additional decisions for the class designer. First, pointers to ordinary objects probably should not be flattened as part of the operator>>= routine of the object unless a copy of the ordinary object is what is desired. Naturally when an eternal object is read back into memory, the ordinary object which existed at the time of the flattening may or may not be around. A decision was made that the pool of eternal objects should be self-consistent thus the use of the normal context mechanism when dealing with ordinary objects was not an appropriate way to guarantee consistency (if we used the normal context mechanism, the objects in the `TPersistentContext` would only be valid under certain conditions).

Pointers to other eternal objects will be handled automatically at flatten and expand time. A reference to these objects will be generated in the stream that you are writing to. At read time, all referenced eternal objects will also be read in. Sometimes the desired behavior is that only the root object is loaded in and referenced objects are loaded only at the time of referencing. The use of a class of smart pointers will solve this problem nicely. Examples of this are given in this document.

MEternal

Eternal objects are just like ordinary objects with the exception that when objects with the `MEternal` mixin are flattened to a stream, only a reference to the `MEternal` object is put on the stream. It is assumed that the `MEternal` object can be loaded during resurrection because the reader of the stream presumably has access to the persistent context (if this is not the case, MEternal objects can be "force flattened" on streams thus guaranteeing they can be reloaded on the backend. Note, of course, that this will not be "the same" object on the backend.). To initially set what persistent context an `MEternal` object is part of, `Add` the object to a `TPersistentContext`. To retrieve an `MEternal` object using its objectid, ask the `TPersistentContext` to retrieve it. To delete the object from a `TPersistentContext`, ask the `TPersistentContext` to delete it.[2] Deleting the object in memory has no effect on the object in the `TPersistentContext`. If a change is made to an `MEternal` object and you would like to have that change reflected in the `TPersistentContext`, two things must be done. First, mark the object as dirty using the `SetDirty` virtual function. Second, either explicitly flush it to the context by calling the virtual function `Add` again or implicitly have it flushed by flattening the object to a stream (or flattening an object that references it).

---

2. Note that no guarantees are made that deleting an object from a TPersistentContext is safe - that is, other objects might reference it and will now have a dangling reference. The problems of garbage collection across pools of persistent objects would be nice but I don't have the time or desire to spend the next ten years of my life on a research project.

```
#define  MEternalMacro() \
          virtual  ObjectID GetObjectID() const                          \
                   { return MEternal::GetObjectID();};                   \
          virtual  void     SetObjectID(ObjectID id)                     \
                   { MEternal::SetObjectID(id);};                        \
          virtual  TPersistentContext*  GetPersistentContext() const     \
                   { return MEternal::GetPersistentContext();};          \
          virtual  void     SetPersistentContext(TPersistentContext* pc) \
                   { MEternal::SetPersistentContext(pc);};               \
          virtual  Boolean  GetDirty() const                             \
                   { return MEternal::GetDirty();};                      \
          virtual  void     SetDirty(Boolean dirty = TRUE)               \
                   { MEternal::SetDirty(dirty);}

class MEternal {
 public:
   virtual ~MEternal();
          ObjectID               GetObjectID() const;
          void                   SetObjectID(ObjectID);
          TPersistentContext*    GetPersistentContext() const;
          void                   SetPersistentContext(TPersistentContext*);
          Boolean                GetDirty() const;
          void                   SetDirty(Boolean dirty = TRUE);
   virtual  TStream&             operator>>=(TStream&);
   virtual  TStream&             operator<<=(TStream&);

 protected:
   MEternal(ObjectID objID = 0, TPersistentContext* context = NIL);
};
```

Note: When subclassing from `MEternal`, include the line:

```
MEternalMacro();
```

in the declaration of your class to automatically have virtual functions in `MCollectible` overridden with the help of the `MEternal` mixin.

TPersistentContext

A `TPersistentContext` is a collection of `MEternal` objects that can be individually accessed and retrieved. `MEternal` objects in one context can reference objects in another persistent context. The retrievel of objects from a persistent context is via an objectid which is automatically assigned at the time of insertion. If it is desired, a mapping could be provided from a "name" to an objectid at a higher level (using the datamanagement tools). Deleting a `TPersistentContext` only removes it from memory. Throwing away the file (the "real" persistent context) will really destroy the persistent context.

```
class TPersistentContext : public MCollectible {
 public:
   TPersistentContext(char* contextName);
   virtual ~TPersistentContext();
   virtual  MCollectible*  Retrieve(ObjectID);
   virtual  void           Add(MCollectible*);
   virtual  void           Remove(ObjectID);
   virtual  void           Delete(ObjectID);
```

```
   virtual  Boolean         IsEqual(const MCollectible* obj);
   virtual  long            Hash();
   virtual  char*           GetName();

   // These routines are used internally by cheetah
   virtual  void            DeferredAdd(MCollectible* objAsEternal, TDeque*);
   virtual  void            CommitDeferredRequest(TEternalWrapper*);
};
```

`TPersistentContext::TPersistentContext(char* contextName)`
Create a new persistent context or open an existing persistent context named contextName.

`TPersistentContext::~TPersistentContext()`
Destroy the object in memory currently that manages the persistent context.

`MCollectible* TPersistentContext::Retrieve(ObjectID objectID)`
Retrieve the object named objectID from the persistent context.

`void TPersistentContext::Add(MCollectible* eternalObject)`
Add (or update) the `eternalObject` passed in. An `objectID` is assigned to the object. Note that passing an object which does not have `MEternal` as a mixin will result in a runtime exception.

`void TPersistentContext::Remove(ObjectID objectID)`
Remove the object named objectID from the persistent context. This is called by the MEternal destructor to clean up the reference in memory. It does not "delete" the object from the persistent context on disk.

`void TPersistentContext::Delete(ObjectID objectID)`
Delete the object named objectID from the persistent context. Remove it from memory and physically delete it from disk.

Example use of Eternal Objects

```
class TArnKey : public MCollectible, public MEternal {
 public:
   TArnKey(const TText& someText, MCollectible* nextOne=NIL);
   ~TArnKey();
   virtual  TStream&        operator>>=(TStream&);
   virtual  TStream&        operator<<=(TStream&);
   virtual  Boolean         IsEqual(const MCollectible* obj);
   virtual  long            Hash();
            const TText&    GetText();
   ... more virtual functions...
   MCollectibleDeclarationsMacro(TArnKey);
   MEternalMacro();
 private:
   TText           fText;
   MCollectible*   fNextOne;
};

MCollectibleDefinitionsMacro(TArnKey);
```

```
TArnKey::TArnKey(const TText& someText, MCollectible* nextOne) :
                                TText(someText)
{
 fNextOne = nextOne;
 SetDirty();
}

TArnKey::~TArnKey()
{
}

TStream& TArnKey::operator>>=(TStream& towhere)
{
 MCollectible::operator>>=(towhere);
 MEternal::operator>>=(towhere);
 fText >>= towhere;
 FlattenPointer(fNextOne, towhere);
 return towhere;
}

TStream& TArnKey::operator<<=(TStream& fromwhere)
{
 MCollectible::operator<<=(fromwhere);
 MEternal::operator<<=(towhere);
 fText <<= fromwhere;
 fNextOne = Resurrect(fromwhere);
 return fromwhere;
}

const TText& TArnKey::GetText()
{
 return fText;
}

Boolean  TArnKey::IsEqual(const MCollectible* obj)
{
 return fText.IsEqual(& ((TArnKey*) obj)->GetText());
}

long TArnKey::Hash()
{
 return fText.Hash();
}

main()
{
 TArnKey* aKey = new TArnKey("hello");
 TArnKey* bKey = new TArnKey("goodbye", aKey);
 TPersistentContext someContext("myContext");

 someContext.Add(aKey);        // aKey added to persistent context. assume
                               // objID = 1 for this example
 someContext.Add(bKey);        // bKey added to persistent context - with
                               // a reference to aKey. assume objID = 2 for
```

```
                                    // this example
  delete aKey;
  delete bKey;                      // Removes in memory versions

  TArnKey* retval2 = someContext.Retrieve(2);
                                    // Loads in both aKey and bKey and restores
                                    // references properly.  Note that aKey was read
                                    // in at this point even though it hasn't
                                    // been referenced in any code.  Look at next
                                    // example to see how fNextOne field could
                                    // be declared in TArnKey to make aKey loaded
                                    // only when referenced.
}
```

And now, another example using a smart pointer to force objects to be demand loaded (rather than all referenced objects being loaded when the "root" object is loaded).

```
class TSmartPointer : public MCollectible {
  public:
    TSmartPointer(ObjectID objID = 0, TPersistentContext* pc = NIL);
    TSmartPointer(MCollectible* realObject = NIL);
            MCollectible*           operator->();
                                    operator MCollectible*();
            void                    operator=(MCollectible*);
    virtual TStream&                operator>>=(TStream&) const;
    virtual TStream&                operator<<=(TStream&);
    virtual ObjectID                GetObjectID() const;
    virtual void                    SetObjectID(ObjectID objID);
    virtual TPersistentContext*     GetPersistentContext() const;
    virtual void                    SetPersistentContext(TPersistentContext*);
            MCollectible*           GetRealObject();
            void                    SetRealObject(MCollectible* realObject);
      MCollectibleDeclarationsMacro(TSmartPointer);

  private:
    MCollectible*         fRealObject;
    ObjectID              fObjectID;
    TPersistentContext*   fPersistentContext;
  };

MCollectibleDefinitionsMacro(TSmartPointer);

TSmartPointer::TSmartPointer(ObjectID objID, TPersistentContext* pc)
{
  fObjectID = objID;
  fPersistentContext = pc;
  fRealObject = NIL;
}

TSmartPointer::TSmartPointer(MCollectible* realObject)
{
  fRealObject = realObject;
  fPersistentContext = fRealObject->GetPersistentContext();
  fObjectID = fRealObject->GetObjectID();
```

```
}

MCollectible* TSmartPointer::operator->()
{
  return GetRealObject();
}

TSmartPointer::operator MCollectible*()
{
  return GetRealObject();
}

void TSmartPointer::operator=(MCollectible* realObject)
{
  SetRealObject(realObject);
}

const unsigned char kNotEternal = 'N';
const unsigned char kYesEternal = 'Y';

TStream& TSmartPointer::operator>>=(TStream& towhere) const
{
  MCollectible::operator>>=(towhere);
  if (fPersistentContext == NIL)
  {
    kNotEternal >>= towhere;
    FlattenPointer(fRealObject, towhere);
  }
  else
  {
    kYesEternal >>= towhere;

    // This forces a flush.
    GetPersistentContext()->Add(fRealObject);

    (GetObjectID()) >>= towhere;
    *(GetPersistentContext()) >>= towhere;
  }
  return towhere;
}

TStream& TSmartPointer::operator<<=(TStream& fromwhere)
{
  MCollectible::operator<<=(fromwhere);
  unsigned char delimiter = 0;
  delimiter <<= fromwhere;
  if (delimiter == kNotEternal)
  {
    fRealObject = Resurrect(fromwhere);
  }
  else
  {
    fObjectID <<= fromwhere;
    fPersistentContext = new TPersistentContext();
```

```
    *fPersistentContext <<= fromwhere;
  }
  return fromwhere;
}

ObjectID TSmartPointer::GetObjectID() const
{
    if ((fRealObject != NIL) && (GetPersistentContext() != NIL))
      return fRealObject->GetObjectID();
    else
    return fObjectID;
}

void TSmartPointer::SetObjectID(ObjectID objID)
{
  fObjectID = objID;
  if ((fRealObject != NIL) && (fRealObject->GetObjectID() != objID))
    {
      OpusBug("setobjectid inconsistency");
    }
}

TPersistentContext* TSmartPointer::GetPersistentContext() const
{
    TPersistentContext* retval = fPersistentContext;
    if (fRealObject != NIL)
    {
      TPersistentContext* aContext = fRealObject->GetPersistentContext();
      if (aContext != NIL)
        retval = aContext;
    }
  return retval;
}

void TSmartPointer::SetPersistentContext(TPersistentContext* pc)
{
  fPersistentContext = pc;
  if ((fRealObject != NIL) && (fRealObject->GetPersistentContext() != pc))
    {
      OpusBug("SetPersistentContext inconsistency");
    }
}

MCollectible* TSmartPointer::GetRealObject()
{
  MCollectible* retval = fRealObject;

  if ((retval == NIL) && (fPersistentContext != NIL))
  {
    retval = fPersistentContext->Retrieve(fObjectID);
      fRealObject = retval;
  }
  return retval;
}
```

```
void TSmartPointer::SetRealObject(MCollectible* realObject)
{
  fRealObject = realObject;
    if (fRealObject != NIL)
    {
    fPersistentContext = fRealObject->GetPersistentContext();
    fObjectID = fRealObject->GetObjectID();
  }
}
```

Now, if TArnKey is declared like:

```
      class TArnKey : public MCollectible, public MEternal {
         public:
            TArnKey(const TText& someText, MCollectible* nextOne=NIL);
            ~TArnKey();
            virtual  TStream&         operator>>=(TStream&);
            virtual  TStream&         operator<<=(TStream&);
            virtual  Boolean          IsEqual(const MCollectible* obj);
            virtual  long             Hash();
                        const TText&   GetText();
            ... more virtual functions...
            MCollectibleMacro(TArnKey);
            MEternalMacro();
         private:
            TText            fText;
            TSmartPointer    fNextOne;
      };
```

the "fNextOne" pointer will automatically demand load the actual object pointed to. Of course, you need to fix up the flatten/unflatten operators to stream fNextOne.

Why do the cheetah classes look the way they do?

There are a number of goals associated with the interface to developers for cheetah capabilities. I shall present them here in no particular order:

1. *Developers should have to write as little boiler plate code as possible (none would be the ideal case).* With proper development system support, all of the boiler plate code could be automatically generated for any of the schemes presented; however, we can not count on this kind of support. Furthermore, the best the development system could provide is a default solution which would be reimplemented in all but the simplest classes.
2. *The interface presented should not add any appreciable overhead to the overall execution of cheetah.* Any interface chosen should not require a significant amount of speed or space overhead to the flattening or expansion of an object. What represents a significant amount of space or speed overhead is certainly open for interpretation.
3. *Semantically different operations on the same class or type should look significantly different.* cheetah provides support for flattening and restoring objects of known type as well as flattening and restoring objects of potentially unknown type. Some believe that flattening an object of unknown type is a very different operations than flattening an object of known type and the developer should probably be aware of this difference. Making the operations look different helps to achieve this goal.
4. *Semantically similar operations should look the same - independent of the class or type the operation is performed on.* Flattening an object of known type should look the same independent of the type the object is. Any object in the system should be able to be flattened the same way.
5. *The actual implementation of the "flatten" and "expand" operations must be symmetric and it should be possible for these implementation to look symmetric.* Without support from a clairvoyant development system, there are two operations that need to be defined for a class: flatten and expand. Flatten and expand are inverse operations and should look symmetric.
6. *The interface should make the invocation of flattening an object and restoring it look "nice" to read. The operations performed should be clear by reading the code.* This is clearly a gray area. Some things look ugly to almost everyone. Some things look nice to almost everyone. Some things look nice to most people but make some people retch violently. I guess I'm hoping we can find something that most (80% ?) people feel comfortable with.

The Ideal Solution

The ideal solution would satisfy the goals above. Ignoring whether or not this solution could be implemented in C++, the code to flatten a group of objects might look something like:

```
{
   TToken          aToken;
   TPoint          aPoint;
   long            aLong;
   char            aChar;
   MResponder*     aResponder;
   ...
   // stuff //
   ...
   TMemoryStream  aStream;
   FlattenObject aToken apoint to aStream;
   FlattenObject aLong to aStream;
   FlattenObject aChar to aStream;
   FlattenPointer aResponder to aStream;
}
```

For now, we'll ignore the actual implementation of the "flatten" methods for a particular class or type. Let's look at the code for expanding a group of objects:

```
{
   // assume aStream is passed in with a flattened set of objects.

   TToken          aToken(aStream);
   TPoint          aPoint(aStream);
   long            aLong(aStream);
   char            aChar(aStream);
   MResponder*     aResponder = UnflattenPointer(aStream);
}
```

Notice that to expand a set of objects, we call a constructor for each type we would like to restore. If we would like to restore an object of unknown type (we know it is an `MResponder` or a class derived from `MResponder`), we call some system function which creates an object of the right type and calls the appropriate constructor.

Without going into the details of how the flatten methods and expand methods (in this example, the expand methods are constructors), this solution certainly satisfies the goal of being somewhat nice to look at. Semantically different operations are distinguishable (`FlattenObject` vs. `FlattenPointer`), and semantically similar operations on different types are performed in the same way. Since I certainly can't implement this in any known language, we can't make any claims about whether this interface adds any overhead to the flattening and unflattening process. It certainly doesn't have to. The flatten and expand methods could be made symmetrical in my mythical language and the developer doesn't have to write any boiler plate code.

Unfortunately, we can't come close to this in C++; however, this so-called "ideal" solution illustrates a number of important features of a good solution in C++:

1. The same operator is used to flatten an object of any type to a stream.
2. A different operator is used when flattening pointers to objects of unknown type.
3. Constructors are used when expanding objects of any type from a stream.
4. The syntax is such that the flattening of multiple objects to a stream could be written as a single statement if that is desired.

Bjarne's Rules of the Game

Before examining the solution space, here is some information about C++ that is absolutely necessary to know when designing a solution to this problem. Reader: I assume that you know C++ already; however, I will point out a number of subtle and not so subtle "features" that are extremely important when designing a solution (After each point, say to yourself, "Thank-you Barney" and bang a large mallet against your head.):

1. C++ provides for constructors of objects only. Built-in types do not *have* to be initialized and there is no syntax that looks like a constructor for initializing built-in types in any event.
2. The operator overloading mechanism in C++ allows binary operations to be overloaded as either member functions taking one argument (the "this" pointer is the first argument) or as a global functions taking two arguments. This is the only mechanism available which allows an expression to be written involving built-in types and user defined types and have the expression look the same independent of the type of the object. An expression, `(a >> b)` will either execute the global function `operator>>(A&, B&)` or the member function `A::operator>>(B&)`. For built-ins, a global functions of two arguments is called. For classes, a member function is called.
3. Constructors have a predefined form which looks like `type-name(args...)`. For example, it is not possible to have a constructor which looks like an operator (Other languages allow any method to be considered a "factory" method used for constructing the object).
4. The syntax for calling superclass and member constructors requires specifying the order and arguments of superclass and member constructors outside the body of a constructor. The syntax for calling superclass and member methods must be inside the body of any other method.
5. Objects have absolutely no idea what they really are only what they currently are. Furthermore, pointers to the same object will be different depending on the current baseclass you are pretending the object is. This means that an object of type C which is descended from classes A and B could be viewed as an A or as a B. A pointer to the object viewed as an A would not be equal to the pointer to the object viewed as a B. This is certainly one of the more disgusting features of the implementation of the language which makes its way up to the semantics of the language.
6. If a class has base classes or member objects with constructors, their constructors are called before the constructor for the derived class. The constructors for base classes are called first. Base classes are initialized in the order specified by the mem-initializer-list in the definition followed by the members in the order specified by the mem-initializer-list in the definition.

Let's Make A Deal

Now that we know the goals, the ideal solution and the limitations, we should be able to examine the alternative solutions. I will try to present alternatives in isolation; however, many choices affect other choices to be made later because C++ is not symmetrical. Let's start with an apparently easy choice first: how to flatten objects.

Flatten

If we want semantically equivalent operations to be expressed identically, independent of the type of the operands, we are left with exactly one choice in C++ on how to do this. We must use operator overloading for flattening objects and built-in types. If we are willing to accept that objects are written to a stream one way and built-ins are written to a stream in a different way, the code for writing an object to a stream would be:

```
anObject->Flatten(aStream);
```

The code for flattening a built-in type to a stream would be:

```
Flatten(aLong, aStream);
```

If we are not willing to accept this, we can easily define a global overloaded operator which takes a built-in type and a `TStream&` and flattens the built-in type. Furthermore, class implementors can define a member operator for a class which can be flattened which takes a `TStream&` as an argument. For the purposes of our discussion, we will assume that the chosen operator is `operator>>>`.[3] We want to be able to make this overloaded operator a member function of various types versus a member function of a stream.[4] Therefore, the C++ statement to write a single object to a stream independent of the type of the object would be something like:

```
anObject >>> aStream;
```

At this point, you're thinking to yourself, that's great. It looks pretty clean. I can live with that. What's the catch? Well, of course, you might like to be able to write expressions like the following:

```
myObjectOfType3 >>> myObjectOfType2 >>> myObjectOfType1 >>> aStream;
```

Actually, you can do this, provided two things hold. First, the overloaded operator that you define must return a `TStream&`. Also, the overloaded operator should group right to left. If you relax the second restriction, you could still write the expression but you'd need to put parenthesis in all the right places as in:

```
(myObjectOfType3 >>> (myObjectOfType2 >>> (myObjectOfType1 >>> aStream)));
```

Forgetting a parenthesis would cause a compile time error which could be easily fixed. Notice that the numbering reflects the order that the object is actually written to the stream. So there are a number of decisions which we would like to make. First: **Should we allow multiple objects to be flattened using an expression which is written in a single statement?** The obvious answer is yes; however, because we'd like to use constructors to expand objects, and constructors are clearly one per statement (and because of other reasons which will be explained later), this question is not so easy to answer. We can enforce the choice that is made by returning or not returning a `TStream&` as the result of the operation.

The next question is: **What operator should we overload for flattening?** Unfortunately, we can't answer this question until we answer the previous question because if we only allow a single object to be flattened per statement then operators which group right to left would not have the obvious advantage.

---

3. This operator does not exist in C++ and therefore could not be used. We will reveal the choices for this operator shortly.
4. If it was a member function of a stream, it would not be extensible in the same way. A global operator (which couldn't be virtual and therefore wouldn't work for our scheme anyway) would have to be defined to flatten your object. This operator would no doubt have to be a friend function so that it could touch your private parts. This is what Barney's stream package does. It's ugly. We're not doing that. We have high standards. We're Apple.

So, let's assume the answer to the previous question and then try to answer this one. If the answer to the previous question is that we will *not* allow multiple flattens in a single statement, then basically any operator is fair game. We'd probably like to choose one that implies some directionality. Therefore our choices are the following:

```
>, >=, >>, >>=, ->, |, ||
```

I can quickly eliminate some of these choices. Using > or >= is unacceptable since they are already overloaded for `MOrderableCollectible` objects. Overloading -> is a very bad idea because it already has a useful meaning for many of the objects we are talking about. Overloading `|` or `||` has the advantage that it resembles a Unix pipe and that is something that C++ programmers are familiar with. Overloading >> certainly has the best directionality; however, it might confuse C++ programmers who use Barney's stream package since the functionality is similar but the syntax would be opposite of Barney's. Overloading >>= has good directionality and it is rarely, if ever, legitimately used; however, it makes atleast one person in Pink violently retch to the point of not being able to think clearly because we are overloading one form of assignment. More importantly, we are overloading assignment in a way that is counter-intuitive since the thing being "assigned into" is on the right hand side rather than the left.

If we want to allow multiple flatten operations in a single statement, we need an operator that groups right to left or we are forced to include parenthesis in the expression. The only choice above that groups right to left is >>=. All the others would require parenthesis in the case where multiple operations are grouped in a single statement.

The next question is how do we flatten an object of unknown type.[5] First, we need to ask the question, **"Is flattening an object of unknown type a significantly different semantic operation than flattening an object of known type?"** Flattening an object of unknown type causes type information to automatically be written out to the stream. When expanding an object of unknown type, it is not possible to use the constructor mechanism. You must use some function with the result cast to some known type (the actual object constructed can be of that type or any derived class from that type). If we believe that flattening an object of unknown type is a semantically different operation than flattening an object of known type then the following might be acceptable syntax:

```
FlattenPointer(myPointerToAnObjectOfUnknownType, aStream);
                              or
myPointerToAnObjectOfUnknownType->FlattenPointer(aStream);
```

What should the name of the function to flatten objects be? It would be desirable for this to be a static member function eventually; however, for now, we could make it a member function if that was desired. If we don't really think this operation is that different even though we accept that expanding an object of unknown type is different then the following would be acceptable and fit right in with flattening objects:

```
myPointerToAnObjectOfUnknownType >>> aStream;
```

Expand

At this point, you are thinking to your self, "Self, these choices weren't so hard. In any event, it looks like things will be pretty clean." Now it is time to start compromising. It is not possible (even if we wanted to) to make the "expand" operation look the same independent of what we are expanding for a number of reasons which will all be explained.

Here is where we need to make our first hard choice. Ideally, one would like to view the process of expanding like a constructor. The argument to the constructor would be a stream. The constructor would then use the stream to build the object. Unfortunately, you cannot call virtual functions in a constructor. This is too great a restriction because some classes require that a virtual function be called to "add" objects to the class when expanding the class from the flattened form. Furthermore, there are no constructors for built-in types.

---

5. When I say unknown type, I really don't mean it. I mean it is definitely of type `MCollectible`; however, we don't know which derived class it is.

Alternatively, we could provide a constructor which builds an empty, uninitialized object (much like an uninitialized built-in), and then call an expand function (in our case an operator so the expressions look the same). So the question is: **"Constructors or operator<<< method?"** The advantage of using an operator is that we could write code like:

```
{
     long    aLong;
     TToken  aToken;
     char    aChar;

     aChar <<< aToken <<< aLong <<< aStream;
}
```

The disadvantage is that we need to provide a distinguished constructor or make the expand method smart enough to deal with an arbitrarily initialized object. (In my example, I've used the constructor with no arguments. We might not want to grab this one because of its legitimate uses.) Furthermore, it is less efficient because now we need to grovel over the structure twice. Once calling constructors and a second time calling inherited expand methods. Additionally, the lack of forced initialization through constructors (which is what we'd essentially provide) for built-in types is a disadvantage.

If we use constructors, we need to still provide some mechanism for reading built-in types from a stream. There are basically two choices. We could just overload the `operator<<<` for built-ins. Good programming style would suggest that you should initialize your variable in the following statement.

```
    {
     long    aLong;
     aLong <<< aStream;

     TToken  aToken(aStream);

     char    aChar;
     aChar <<< aStream;
}
```

Another option is defining conversion operators (see Bjarne's book page 174) for all the built-in types which convert streams to built-in types using an operation that looks like a constructor. The disadvantage is that this is a slimy thing to do that could bite us in the butt sometime down the road. If we did this slimy thing (which I have to admit has a great deal of appeal to me), the code snippet to read in our things is:

```
    {
     long    aLong = long(aStream);
     TToken  aToken(aStream);
     char    aChar = char(aStream);
    }
```

As usual, there are two choices for how to read back unknown objects. We could provide an overloaded operator to read objects back in. However, in order to implement this, we would be forced to change the runtime of C++ to use a scheme which did not move the "this" pointer in an object.[6] I will assume that this is not practical. Therefore, the only choice is to provide some kind of function which performs the "new" of the object and calls the constructor. The user will have to write the cast back. Therefore, the code to bring back an `MResponder*` of unknown derived type would look like:

```
          MResponder* aResponder = (MResponder*) Resurrect(aStream);
```

6. You'll have to accept my word on this. Basically, the problem is that we need to do a cast back when reading in an object of unknown type (actually an `MCollectible*`) if we want to store this object as anything other than an `MCollectible*`. The only way to write this cast back is to actually write it in the code as part of an assignment statement. If we changed the runtime, this cast back would be unnecessary.

What should the name of global function to "resurrect" an object be? When there are static member functions, this can be a static member of `MCollectible`. For now that is not possible (because you would need one `MCollectible` object to resurrect another one). Because this is our only choice for expanding an object of unknown type (which many consider to be a semantically different operation than expanding an object of known type) and our desire for symmetry between flatten methods and expand methods, we should carefully examine our decision for how we want flattening unknown objects to look.

Implementation of operator<<< and operator>>>

All user defined types which want to be flattened to a stream to be restored later must implement a operator<<< which takes a `TStream&` as an argument. Also, they need to implement `operator>>>` which also take `TStream&` as an argument (and may also return this depending on whether we allow more than one flatten per statement). Classes which want to be flattened to streams via a pointer to an unknown type, must descend from `MCollectible`.[7]

Let's take a look at a few examples:

```
class TPoint {
 private:
   long  fX;
   long  fY;
 public:
   TPoint();
   ...other constructors
   TStream& operator<<<(TStream&);
   TStream& operator>>>(TStream&);
   ...other member functions
};


TStream& TPoint::operator<<<(TStream& aStream)

{
   fX <<< aStream;
   fY <<< aStream;
   return aStream;
}

TStream& TPoint::operator>>>(TStream& aStream)
{
   fX >>> aStream;
   fY >>> aStream;
   return aStream;
}
```

This example illustrates that adding these methods to a class without a vtable does not force this class to have a vtable and thus adds no overhead to the size of the objects. Reading and writing points to a stream is trivial as in the following example:

```
{
   // Writing a point
   TPoint   aPoint(100,200);
```

7. This will make the `operator>>>` a virtual operator. Also, there is one other method which must be overridden (to supply the class name) but that is unimportant for our discussion.

```
   aPoint >>> aStream;
}

{
   // Reading a point
   TPoint   aPoint;
   aPoint <<< aStream;
}
```

Okay, let's look at the implementation of the flatten and expand operations for a slightly more complicated class:

```
// TView's lineage is not entirely known; however, we do know that it
// is an MCollectible object.  This means that there are additional
// methods that we need to implement but they are not important for
// this example (the additional method is GetClassNameAsString.  It
// could easily be optimally generated by HOOPS.)

class TScrollbar : public TView {
 private:
   TPoint   fThumbPosition;
 public:
   TScrollbar();
   ... other constructors
   TStream& operator<<<(TStream&);
   TStream& operator>>>(TStream&);
   ... other member functions
};

TStream& TScrollbar::operator<<<(TStream& aStream)
{
   TView::operator<<<(aStream);
   fThumbPosition <<< aStream;
   return aStream;
}

TStream& TScrollbar::operator>>>(TStream& aStream)
{
   TView::operator>>>(aStream);
   fThumbPosition >>> aStream;
   return aStream;
}
```

Now flattening and expanding `TScrollbar` objects looks identical to all of the other examples. Note that the order that objects are flattened should follow the order that they are expanded.

Writing or reading an object that is known to be a `TScrollbar` but could be a subclass of `TScrollbar` is accomplished with the following:

```
{
   // Write out a TScrollbar*
   TScrollbar* foo;
   ... got foo from somewhere.
   FlattenPointer(foo, aStream);
}
```

```
{
   // Read in a TScrollbar
   TScrollbar* aScrollBar = (TScrollbar*) Resurrect(aStream);
}
```
As a final example, let's examine a somewhat complicated class:
```
class MResponder: public MCollectible, public MCollectible {
 private:
   TToken          fInstanceName;
   MResponder*     fSuperTarget;
 protected:
   MResponder();
   ... more constructors
 public:
   TStream& operator<<<(TStream&);
   TStream& operator>>>(TStream&);
};

TStream& MResponder::operator<<<(TStream& aStream)
{
 MCollectible::operator<<<(aStream);
 fInstanceName <<< aStream;
 fSuperTarget = (MResponder*) Resurrect(aStream);
 return aStream;
}

TStream& MResponder::operator>>>(TStream& aStream)
{
 MCollectible::operator>>>(aStream);
 fInstanceName >>> aStream;
 FlattenPointer(fSuperTarget, aStream);
 return aStream;
}
```
Expanding an `MResponder` from a stream is *not* possible. Expanding a derived of `MResponder` from a stream or "resurrecting" an `MResponder` from a stream is possible. This is because the constructor is protected.

Conclusion

C++ is an ugly language. The design of the operating system of the 90's is being driven by a short, balding, Scandinavian dude currently residing in New Jersey and working for the the phone company.

I really want to know what you think about what was presented here or maybe you have another scheme entirely. In any event, my current mode of thinking is to make the following choices:

1. Allow multiple flattens in a single statement; however, discourage its use when writing flatten methods because if you write your own flatten methods like this then it looks less symmetric with the constructor. I've waffled on this a bit; however, I think it is legitimate to want to write multiple flattens in a single statement when flattening objects in the rest of your code. It's quite possible that symmetry is impossible in these cases anyway since another module, program, etc. could be the one reading it back in. In that case, you might provide a specification of the format rather than an actual piece of code. I don't know. I still need your help.
2. Since I'm still waffling on number one, I'm still waffling on the operator to use. I have to admit, I'm favoring `operator>>=` and `operator<<=` even though it makes one person violently ill. I don't really like any of the other choices. I'm down on using >> because of its current meaning in C++

which is too close to how we want to use it.

3. Use expand methods because you can't call virtual functions in constructors.
4. Encourage the use of `FlattenPointer` which will be a static member function eventually.
5. The name of the function to bring back an object is `Resurrect`.

56

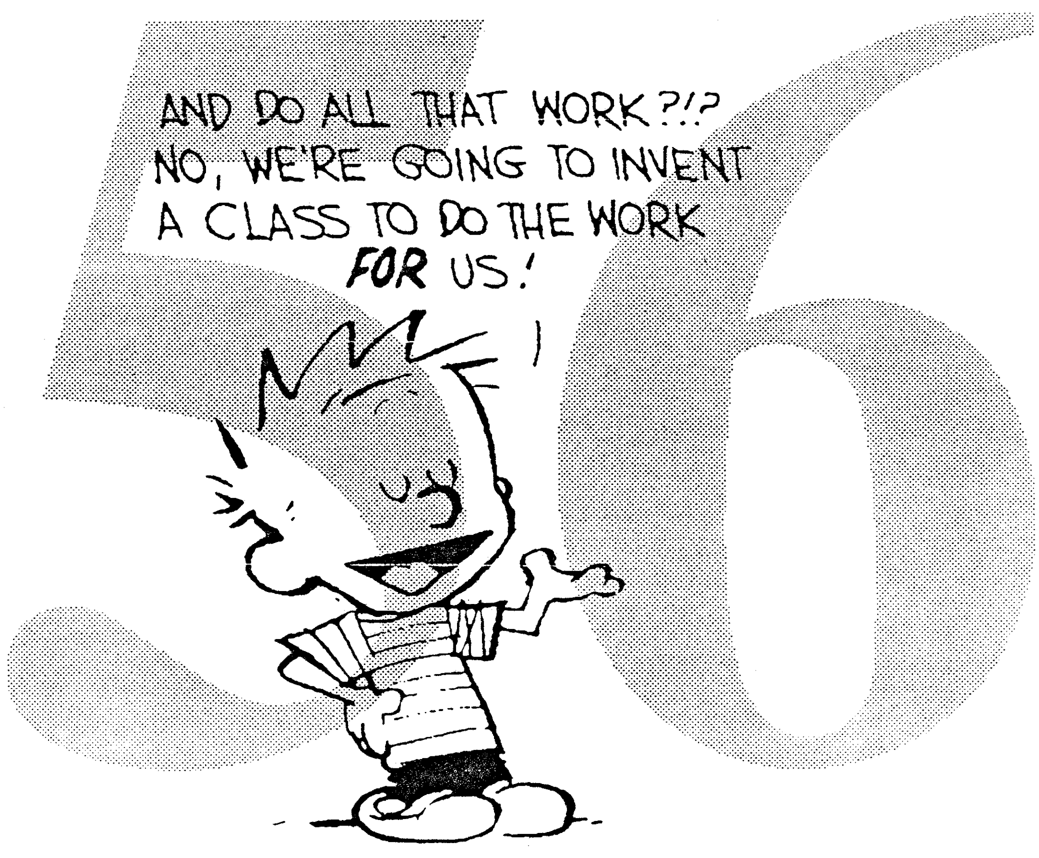
Concurrency Control & Recovery

56

Credence ERS

Pink CCR Classes

Charles Niemeier
x49209



56

Introduction

Credence[1] provides concurrency control and recovery (commonly called CCR) for the Pink environment. Concurrency control handles the problem of multiple tasks accessing data at the same time by making it appear, at least to your task, as if no sharing is occurring. Recovery addresses the problem of failures that leave data in an inconsistent state by making it appear that failures never occur. Failures, in this paper, include program, power and system failures but do not include media failures. Most existing CCR software have been components of specific programs, e.g. a DBMS, for which the CCR design has been tailored. Credence is a set of classes and protocols that offers CCR functionality for the extensible data types possible in the Pink world.

Who Needs This Stuff?

You need concurrency control if you have more than one task , either in the same team or in different teams, accessing the same data and at least one task is changing the data. You need recovery if you wish data to survive failures and remain logically consistent. Concurrency control and recovery is not automatic. It takes effort on the part of a class designer to use Credence properly. Hopefully, Credence provides a framework that is useful enough to discourage the need for ad hoc concurrency control and recovery schemes.

CCR Terminology

Concurrency Control

You need concurrency control when data is shared. You are probably familiar with complications caused by reentrancy in a multithreaded environment. Common techniques to manage concurrency usually involve some form of mutual exclusion. For instance, you can use semaphores (see Pink Runtime) to obtain mutual exclusion. Mutual exclusion controls concurrency by forcing serial execution, i.e. concurrency is suspended for some time. It is possible to perform certain operations concurrently while producing the same results as a serial execution. An execution that produces a result equivalent to a serial execution is called serializable. Some applications can use the definition of serializability to control sharing while resulting in more concurrency than with mutual exclusion methods.

The most common techniques for concurrency control involve acquiring and releasing locks of some type. A lock is an abstraction such that when you acquire (or set) it, you have access to some associated object and prevent others from accessing the object until you release the lock. A task, which is requesting a lock, is blocked until the task gets the lock. Locks usually have a type, e.g. read and write locks, that reflects which operation you are about to do. Depending on the type of the lock, multiple users may be able to concurrently acquire a lock and still preserve serializability.

---

[1] Although the etymology is obvious, convention was elected over Mr. Fogerty's iconoclastic spelling.

Serializability and locks have their limitations. For some applications, serializability may not be desired. The syntactic notion of holding locks also may be too restrictive. For instance, an operation that balances a tree does not change the semantic content of the tree but a lock on a node or branch of the tree might prevent the tree balancing operation.

You might still be asking, "But what's the difference between locks and semaphores?" Actually, locks are very similar to sharing semaphores but provide the extra baggage to release locks in a manner that guarantees recoverability and to handle deadlocks.

Recovery

Failures can can cause logical inconsistencies in data. These inconsistencies can result when two or more data items, all dependent on each other in some way, must be written to disk but the disk hardware can not write all items in one fell swoop. Recovery is a guarantee that failures don't result in data inconsistencies on disk.

Transactions

A transaction is a useful concept for dealing with recoverability and concurrency control. A transaction is a sequence of operations with the following characteristics:

- A transaction is consistent, which means that it either succeeds and commits or aborts and has no effect.
- A transaction is persistent, meaning that the results of a completed transaction should survive failures.

I'll abbreviate the consistent and persistent definition of transactions by saying that a transaction is atomic.[2]

In terms of what I've already presented, a transaction is that set of operations that should look serial relative to the operations of some other transaction or a transaction is that set of disk updates that must all succeed.

Recoverability

Recovery is related to concurrency control as seen in the definition of recoverability:

- A transaction T1 is said to read from transaction T2 if T2 writes some x which is later read by T1.
- A transaction T1 is recoverable iff T1 commits after every transaction from which T1 read has committed.

[2]My definition of atomic differs slightly from that in most database CCR literature which also includes serializability as a condition for atomicity.

The requirement that a transaction commits after every transaction from which it has read is sufficient for recoverability but can lead to cascading aborts. We can make our implementation of recovery easier by requiring: 1) that a transaction only reads from committed transactions; and 2) that a transaction can only write x after every transaction that previously wrote x has committed. The first requirement avoids cascading aborts and the second allows before images to be used in the log. An execution that meets these two additional requirements is said to be *strict*.

Two-Phase Locking

Two-Phase Locking (2PL) is a protocol for using locks that provides serializable concurrency control and guarantees strictness. The 2PL protocol involves acquiring and releasing locks in two phases: locks are acquired during the course of a transaction (the first phase) and released, all at once, at the end of the transaction (the second phase).

Logical Locks

In the database world, locks are usually associated with known objects, e.g. records. How can we use locks with an extensible type system? One approach is to use logical locks. A logical lock is, basically, just an integer value.

The use of logical locks is left to your creativity. For example, you can place locks on individual data items by using their offsets in the file. Alternatively, you could use special values to indicate a lock on a set of data items, for instance, blocks in a file or an entire file.

Distributed Transactions

A transaction system is distributed if a transaction can involve work on more than one machine in a network. Distributed transaction systems are more complex than the non-distributed model because they need to deal with events like the failure of some of the machines while other machines remain operational and partitioning of the network. Atomic commit protocols have been developed that try to insure that a transaction will only commit if all nodes can commit.

Credence does not provide a distributed transaction facility. This means that work can't be done on multiple machines for the same transaction.

The most important consequence of the lack of distributed transactions is that certain operations might act differently for local files than for files on Appleshare volumes. For instance, we expect the Pink file system server to work with the transaction system so that a file system request made from within a transaction, for instance, truncating a file, would only truly truncate the file when the transaction commits. This might not be possible with network files.

Architecture

Credence provides classes that provide transactions, recovery, and concurrency control. Classes are available to create transactions, change files in a recoverable manner, and request 2PL locks. Moreover, Credence provides a framework for recovery and concurrency control that allows users to modify the default mechanisms without starting from scratch.

Transactions

Credence provides a class (TTransaction) that allows you to start and end transactions within a task. There is, at most, one transaction for a task and the transaction is only active for the task that started it. Methods are available to commit transactions, abort transactions, and find which transaction, if any, is currently active for the task.

The definition of a transaction involves control flow because, when a transaction aborts, you typically don't want to perform the remaining operations in the transaction. For this reason, the transaction class uses C++ exceptions to alter the control flow when a transaction aborts and defines a protocol for handling these exceptions.

Concurrency Control

Credence concurrency control is based on locks. A central Lock Manager team services all lock requests for the local system. A concrete class (TLogicalLock) is provided that offers 2PL locking based on logical locks. When you request the lock, the request is sent to the Lock Manager and your task will block until the lock is acquired. Because these are 2PL locks, you never explicitly release a lock because locks are automatically released at the termination of the transaction. The Lock Manager performs deadlock prevention and may abort a transaction because of potential deadlock.

A logical lock is composed of three components: an unsigned long value , a type, and a file specifier. You can use the value of the lock to mean whatever you like, as discussed earlier. The type of a lock is either read or write. A task will block if it tries to set a lock that conflicts with an existing lock. Locks conflict if they have the same value and file, different transactions are requesting them, and their types conflict. For read and write locks: read locks conflict with write locks; write locks conflict with both read and write locks. You use the file specifier so that a lock on one file does not conflict with a lock of the same value on another file. For example, placing lock 0x100 for file "Pink" should not conflict with lock 0x100 for "Blue".The file specifier is used to identify for which file the lock is set.

You can, however, use locks other than logical locks. An abstract superclass (TLock) exists from which you can derive new lock subclasses. The Lock Manager handles these extended lock classes by using lock controller objects (TLockController). Every lock subclass must have a corresponding lock controller subclass. When the Lock Manger receives a lock request, the Lock Manager passes the lock object to the appropriate lock controller.

Because of the overhead involved, you should not use locks when semaphores will do.

Recovery

Credence provides classes that let you easily access files in a recoverable manner. One concrete class (TSafeFileSegment) exists that allows you to use memory-mapped files in a normal fashion and makes changes recoverable with a minimum amount of work. Another concrete class (TSafeServer) provides a recoverable stream interface to a file that is handled by a separate team, a data server. You use a data server when you want to share the file between more than one team, or you don't want to map the file into your address space, or the file is larger than a team's address space (because the data server will use a pool of buffers to access the file).

Credence recovery uses write-ahead logging in which all changes to recoverable objects are first written in a central, system log. The Credence system provides the facility for recording changes into the system log and playing back the lock in the event that a transaction aborts or a system failure occurs. The default type of logging is called physical logging because images of which bytes changed are saved in the log. Credence Recovery is based on two abstract classes: MRecoverable and TLogRecord.

Any object that is considered recoverable (a disk file) must be an MRecoverable. Any changes to an MRecoverable results in a log record. The MRecoverable class has a method that outputs the log record to the log file which must be overridden, thus allowing a subclass to do its own logging. MRecoverable hooks itself into the transaction system so that the transaction system can notify it of events. MRecoverable has methods that are called when transactions start, commit, and abort since this information is needed if the class is doing its own logging and can also be useful for other purposes. MSystemRecoverable is a subclass of MRecoverable that uses the central, system log. The MSystemRecoverable class and the underlying transaction/log framework is flexible enough to accomodate undo/redo, undo/no-redo, and no-undo/redo recovery schemes.[3]

The type of log record used with an MRecoverable can take many different forms depending on what algorithms are used for logging. For instance, some logging schemes use logical log records while other schemes use physical log records; some log records include before-images, after-images, or both. Log records are created by instantiating objects of some subclass of TLogRecord.

The transactional system uses information in the log under two circumstances — when a transaction aborts and after a system failure (Restart). In both of these conditions, the process that created the log record may not be running anymore. Changes made by aborted transactions may need to be undone and changes by committed transactions may need to be redone. The transaction system can not know how to interpret every type of log record. Each TLogRecord subclass, therefore, has methods to redo or undo the change represented in the log record. These methods could redo or undo the changes themselves or they could send a request to some team in the system.

TSystemLogRecord is a subclass of TLogRecord that you use to instantiate log records that will work in the central, system log. TSystemLogRecord includes information that the system logger needs to work properly.

Figure I shows the teams that make up the Credence system and a simplified message flow. Also shown (in boxes) are the concrete classes that send these messages.

---

[3]The names of these recovery schemes come from whether you need to undo changes by aborted transactions or redo changes made by committed transactions in order to recover.

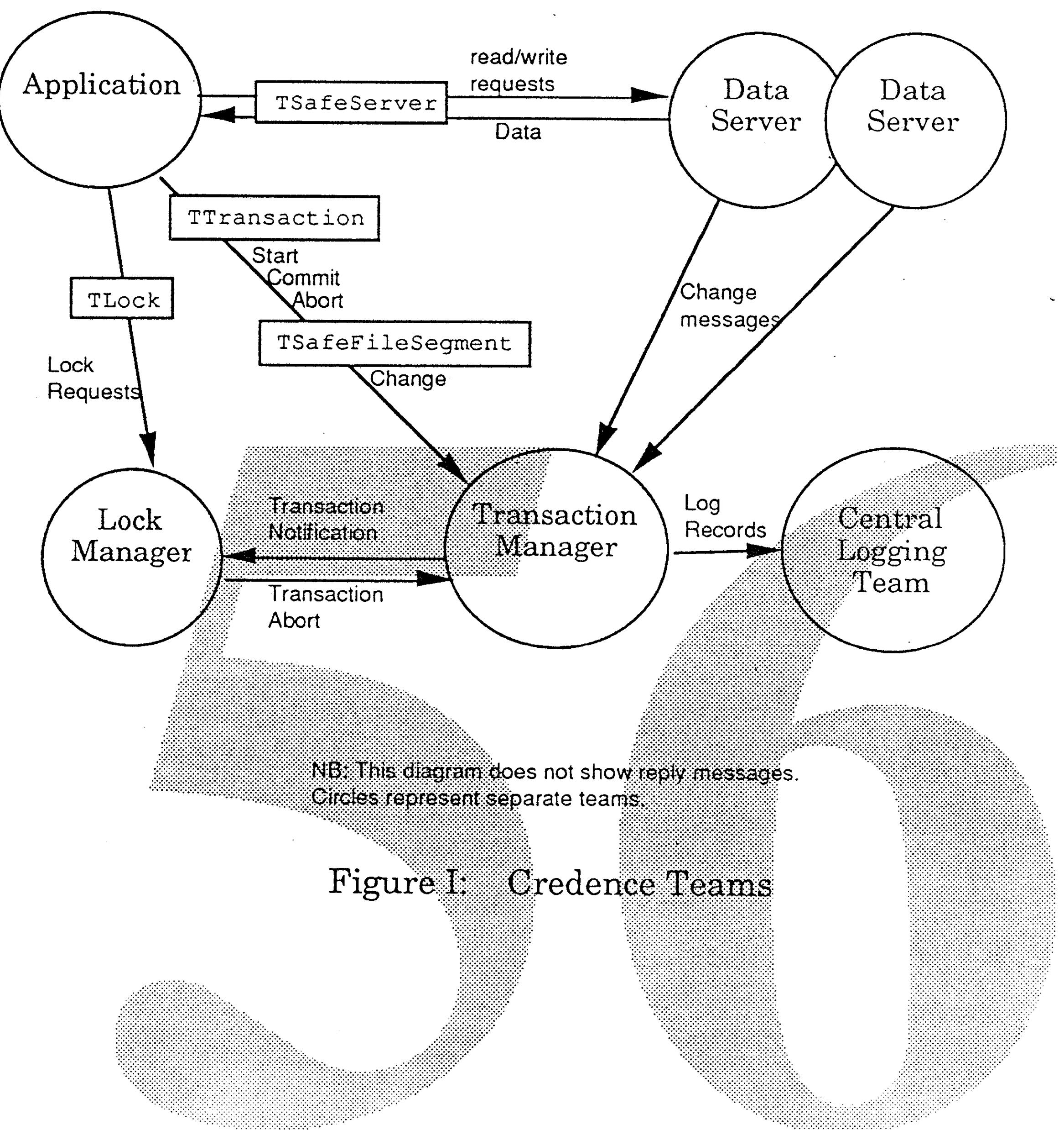
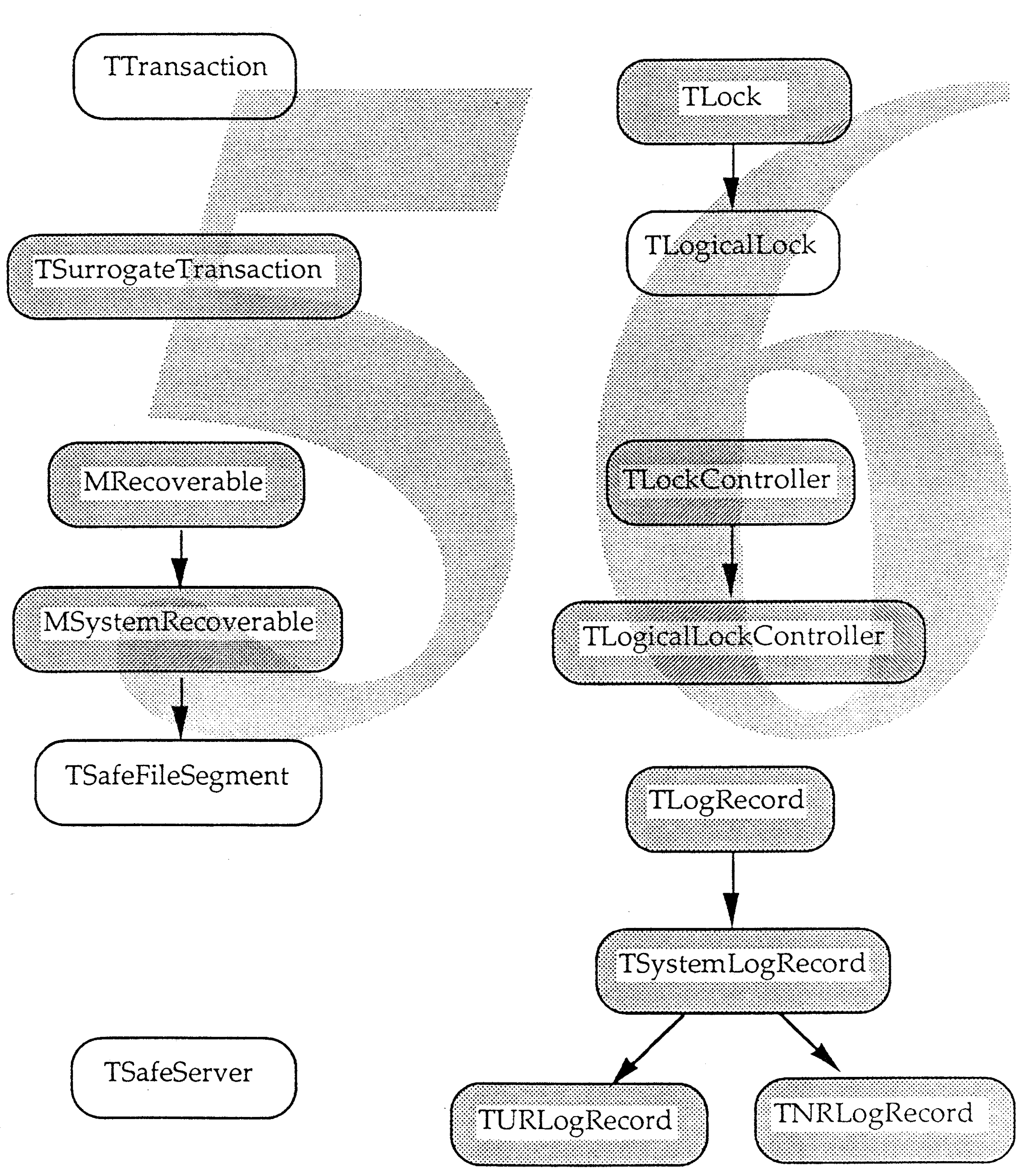


Figure I: Credence Teams

Class Diagram



Classes and Methods

TTransaction

A transaction is started by the definition of a TTransaction object. The most important methods that you will use are the following:

- TTransaction – the constructor starts the transaction
- Commit – commits the transaction.
- Abort – aborts the transaction.
- GetActiveTransaction - returns a pointer for the TTransaction active for the current task.

An exception is raised if the transaction aborts. For this reason, code that involves transactions should be prepared to catch exceptions. Until we have real C++ exceptions, we need to use Andy's exceptions (see Pink Runtime). The following code illustrates creating and committing a transaction along with the exception code.

```
foo()
{
        TTransaction t;

        TRY
                // transactional operations

                t.Commit();
        EXCEPT
                // put code here to execute if the transaction aborts
        ENDTRY
}
```

You commit a transaction by calling Commit. The destruction of a TTransaction aborts the transaction if Commit hasn't been called. You can also explicitly abort a transaction by using Abort.

Only one transaction can be active for a task at any one time. An attempt to instantiate a TTransaction when a transaction is already active will raise an exception. If you won't know whether a transaction might be active entering a function, you can use GetActiveTransaction to find out.

A Preview of Exceptions

The design of C++ exceptions still hasn't stabilized but the Credence classes will use something similar to the following (based on Stroustrup's design at the time I wrote this paper).

There will probably be at least one class, TTransactionException, of which an instance is used to signal an exception associated with transaction errors. The following example shows how you would use transactions with exceptions.

```
foo()
{
        TTransaction t;
```

```
	try
	{
		// transactional operations

		t.Commit();
	}
	catch( TTransactionException p )
	{
		// put code here to execute if the transaction aborts
	}
}
```

TSafeFileSegment

The TSafeFileSegment class is used much like a TFileSegment except that changes are atomic. You can access a TSafeFileSegment directly or through streaming operations. Any direct change must be bracketed by calls to ChangingContents and ChangesDone. Any block of contiguous operations that write to the stream must be bracketed by calls to the ChangingStream and ChangesDone functions.

Examples.

```
	// change 100 bytes within a memory-mapped file
	TSafeFileSegment myFile(filePtr);	// filePtr is a TFile*
	myFile.ChangingContents(destination, 100);
	memmove(destination, source, 100);
	myFile.ChangesDone();

	// make some changes through the TStream interface
	myFile.ChangingStream();
	x >>=  myFile;
	y >>= myFile;
	z >>= myfile;
	myFile.ChangesDone();
```

One of the concurrency control responsibilities that you have when accessing a TStream from more than one task is to assure mutual exclusion when you stream contiguous blocks from the stream. For a TSafeFileSegment, the ChangingStream and ChangesDone functions also need to be within this mutual exclusion interval. For better concurrency on the permanent segment, use more than one TSafeFileSegment and Credence concurrency control, e.g. TLogicalLock.

Executions defined by ChangingContents (or ChangingStream) and ChangesDone pairs must be disjoint, i.e. a ChangingContents (ChangingStream) must be followed by a ChangesDone before you can invoke another ChangingContents (ChangingStream).

TSafeServer

The TSafeServer allows you to access a file atomically through a data server. You can use the TSafeServer class when you need to share a recoverable object between teams or you don't want to map a file into your application's address space. The methods of this class will send read and write requests to a data server. All access to the file is via TStream operations. You use the ChangingStream and ChangesDone functions in the same way as for TSafeFileSegment. The server team is launched if one doesn't already exist for the file. The server team will use a pool of segments to access the file instead of mapping the file completely into the server's address space if the file is larger than a team's address space.

The data server does not automatically perform concurrency control. However, you can use TLogicalLock to control concurrent access to a data server. De-coupling locking from data servers allows greater flexibility in locking while allowing simpler data servers.

Note. I could design a couple more types of servers if demand warrants. For instance, a server that only handles write requests because the clients have a read-only, shared segment in their own address space.

TLock and TLogicalLock

The TLock class is an abstract class that provides the framework for lock-based concurrency control. The most useful methods of this class are as follows:

- SetLock – set (or acquire) the lock.
- Release – release the lock.

The TLogicalLock class is a concrete class derived from TLock that provides 2PL logical locks. As mentioned earlier, logical locks have 3 components: a file specifier, lock value, and type. These components can appear as arguments to the constructor of the TLogicalLock or can be accessed with getter and setter functions. The use of a logical lock is shown in the following code:

```
foo(const TFile& file)
{
        TTransaction t;
        // set a read lock with value 100
        TLogicalLock lock(file, 100L, TLogicalLock::kReadLock);
        TRY
                lock.SetLock();         // set the lock
                // do something
                t.Commit();
        EXCEPT
        ENDTRY
}
```

Notice that a TLogicalLock never should call Release explicitly since a 2PL lock is automatically released at the time the transaction terminates.

TSurrogateTransaction

When a TTransaction is instantiated, the object is associated with the task that is running and communication is set up to the Transaction Manager if such communication hasn't been established already. The Credence teams, Logger and Lock Manager, deal with these transactions but don"t want to do the actions mentioned above each time they resurrect a transaction object. The TSurrogateTransaction class is used for this reason to provide surrogates for transactions.

TLockController and TLogicalLockController

When a TLock is set, it actually sends a request to the global Lock Manger team. The Lock Manager determines which subclass of TLockController is needed to control requests for the lock being requested and instantiates a TLockController if one doesn't already exist. TLogicalLockController is, naturally, the subclass that handles TLogicalLock requests.

MRecoverable and MSystemRecoverable

As mentioned earlier, MRecoverable is a class that you must mix into any object to which you want to apply recoverable changes. It has the knowledge to hook into the transaction system so that it can find out about transaction events. You can obtain complete control over logging by deriving from MRecoverable. The MSystemRecoverable class is derived from MRecoverable. MSystemRecoverable knows how to communicate with the transaction system and the system logging process.

The abstract MRecoverable class doesn't have many methods - much of its work has now been taken up by the TLogRecord class. TSystemLogRecord is a subclass of TLogRecord that includes the information the system logger needs.

The MSystemRecoverable class has two methods, Checkpoint and ForceLog, that require some explanation. Most implementations of a log need to checkpoint the log occasionally for performance reasons. To do a checkpoint, the log needs to know that changes made to recoverable objects have been flushed to disk. The purpose of the Checkpoint method is to tell an MSystemRecoverable that any changes need to be forced out to disk. Also, when log records are sent to the system logger, they are buffered to reduce the amount of I/O. However, you must not write changes made by uncommitted transactions to disk until after the log records are safely on disk. For instance, if the virtual memory system wants to page out an area that was changed by a transaction that hasn't yet committed, we must be sure that the associated log records are written to disk first. The ForceLog method causes the system to force the log buffer to disk.

TLogRecord and TSystemLogRecord

TLogRecord is the abstract superclass of all log records. If an MRecoverable method changes an MRecoverable, the method is responsible for instantiating the right kind of TLogRecord for the logging scheme in use for that MRecoverable. The only methods of TLogRecord are as follows:

- Undo – undo the changes represented in this log record.
- Redo – redo the changes represented in this log record.

TSystemLogRecord is an abstract class that includes enough information in the log record to work with the system log.

TURLogRecord and TNRLogRecord

The TURLogRecord and TNRLogRecord classes are concrete subclasses of TSystemLogRecord that support undo/redo and no-undo/redo recovery, respectively. These are used within the transaction/logging system to implement TSafeFileSegment and TSafeServer.

More Things You Need to Know

Classes that use locks must be good citizens. You must make sure that all classes accessing shared data use a common locking scheme. You can use locks in concurrent tasks in a team to access data in memory. You also use locks by multiple teams to control access to data in a TSafeServer.

Anyone using MSystemRecoverable must ensure that the executions are strict. Executions will be strict if one of the following cases is true:

- The data is not shared.
- The data is not used concurrently, i.e. access is serialized.
- You use the concurrency control of TLogicalLock.
- You guarantee strictness with your own concurrency control.

The transactional operations you include in recoverable objects should only be operations that are necessary for logical consistency of the data. Some operations might result in an inordinate amount of log I/O and are not necessary for logical consistency. For example, assume that you copy a file within a transaction. This would result in writing the entire file twice, once into the log and again into the destination file.

Examples

The first example shows how the locking would work for classic database concurrency control.

```
// Set lock for record "fred" in the file "My File"
//
foo(const TFile& file)
{
	// first, create a lock object (doesn't set it yet)
	TLogicalLock myLock(file, 0, kWriteLock);

	// Use a data base index object capable of mapping a key to a
	// record ID value.  The record ID will be used as the lock
	// value in the logical lock
	long value = index.MapToID("fred");
	myLock.SetLockValue(value);

	TTransaction t;				// create the transaction
	TRY
		// now set the lock
		myLock.SetLock();		// This may block until the lock can be set

		// do something to the record

		t.Commit();			// commit the transaction
	}
	EXCEPT
		error("transaction aborted unexpectedly!");
	ENDTRY
}

// Set lock on an entire file
//
const long kWholeFile = 0L;

f(const TFile& file)
{
	TTransaction t;
	TLogicalLock wholeFileLock(file, kWholeFile, kWriteLock);

	TRY
		wholeFileLock.SetLock();
		//
		t.Commit();
	EXCEPT
		error("transaction aborted unexpectedly!");
	ENDTRY
}
```

The next example shows a member function of some random class that writes to a TSafeFileSegment.

```
TRandomClass::WriteRecoverably(TSafeFileSegment& stream)
{
	TTransaction t;
```

```
	TRY
		stream.ChangingStream();
		fFoo >>= stream;
		fBaz >>= stream;
		stream.ChangesDone();
		t.Commit();
	EXCEPT
		qprintf("transaction was aborted\n");
	ENDTRY
}
```

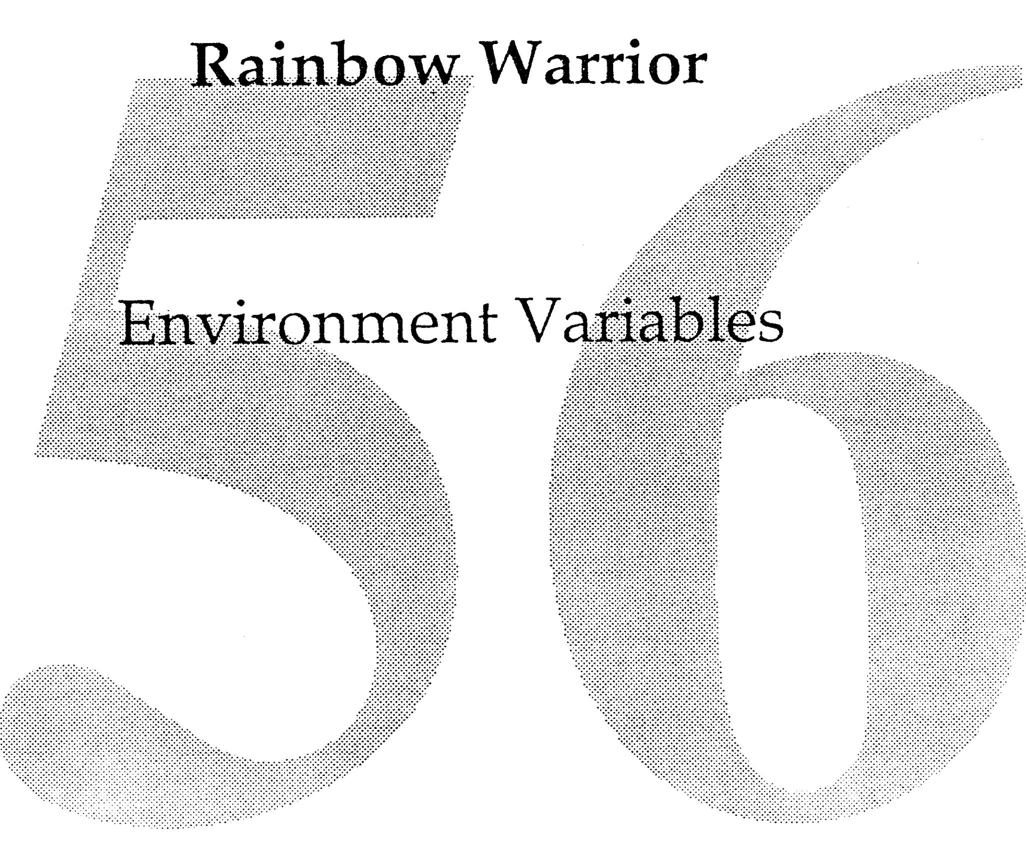
The next example shows writing to a TSafeServer. (Notice that the operations are identical to the stream operations using a TSafeFileSegment.)

```
TRandomClass::ServerWriteRecoverably(TSafeServer& server)
{
	TTransaction t;

	TRY
		server.ChangingStream();
		fFoo >>= server;
		fBaz >>= server;
		server.ChangesDone();
		t.Commit();
	EXCEPT
		qprintf("transaction was aborted\n");
	ENDTRY
}
```

The following example aborts the current transaction.

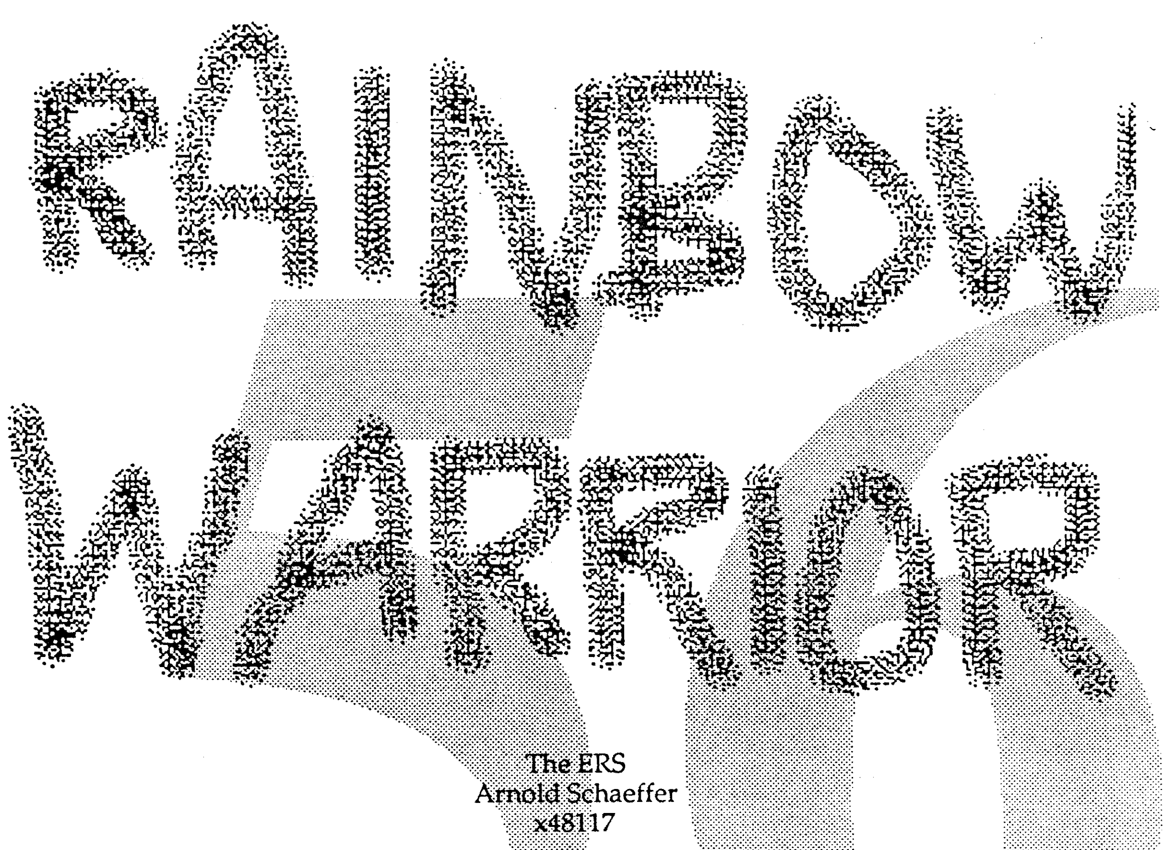
```
void Cancel()
{
	TTransaction *const xactPtr = TTransaction::GetActiveTransaction();
	if(xactPtr != NIL) {
		TRY
			xactPtr->Abort();
		EXCEPT
			qprintf("yes, it did abort\n");
		ENDTRY
	}
}
```

Rainbow Warrior



Environment Variables

56



The ERS
Arnold Schaeffer
x48117

The purpose of Rainbow Warrior is to provide an "environment variable" registry and notification system under Pink. The combination of Rainbow Warrior, Psycho Killer and the "read-only" resource fork provides a superset of the services available with the Blue resource manager.

56

Introduction

The purpose of Rainbow Warrior is to provide an "environment variable" registry and notification system under Pink. The combination of Rainbow Warrior and the "read-only" resource fork provides a superset of the services available with the Blue resource manager.

Rainbow Warrior provides facilities for accessing and updating environment variables (both system environment variables and local environment variables). It also provides facilities for receiving notification when a change is made to an individual variable or a change is made to any variable belonging to a particular category. For example, applications using this facility can be notified when new fonts are installed in the system or additional shared libraries are available. Rainbow Warrior provides a stack of environment variables "environments" so applications can shadow variables declared in the global environment with their own definitions. Finally, clients can enumerate all of the environment variables or all of the environment variables in a particular category.

A mechanism for pre-change notification and "voting" for a change to be allowed is being considered.

NOTE: This is a very preliminary ERS. There is an example provided and "headers" for the classes that a client would use.

Example

```
// For the purpose of this example, assume that a method of a TApplication is
// GetEnvironmentStack which returns the current "environment." There is also
// a call in TApplication which returns the application's "TEnvironment."

TEnvironmentStack* environment = myApp.GetEnvironmentStack();
TEnvironment* applicationEnvironment = myApp.GetApplicationEnvironment();

// applicationEnvironment is already part of the environment stack.

// Query for a particular shared library
TToken category("SharedLibrary");
TToken instance("SpecialTextClasses");
TSharedLibrary* lib = (TSharedLibrary*)
                          environment->Retrieve(category, instance);

// Test for the existence of a particular shared library
Boolean exists = environment->Member(category, instance);

// Check for the user name
TToken userNameCategory("User Name");

// The returned TToken object belongs to you.
TText* userName = (TText*) environment->Retrieve(userNameCategory);

// Add an environment variable to the "application environment." This shadows
// any definition of the same environment variable found in another
// "TEnvironment" further in the "environment stack." Note that there are
// atleast two other "environments" below the "application environment" in the
// environment stack: the "user environment" and the "system environment."
TToken userShoeSize("Shoe Size");
TCollectibleLong shoeSize(9);
applicationEnvironment->Add(userShoeSize, &shoeSize);
```

```
// Add an environment variable that is has a category and instance name
TToken category("SharedLibrary");
TToken instance("MySharedLibrary");
TSharedLibrary* lib = new TSharedLibrary(....);
applicationEnvironment->Add(category, instance, lib);

// Get an iterator to iterate over all of the sharedLibraries.
TIterator* anIterator = environment->CreateIterator("SharedLibrary");
TToken* nextLibrary = (TToken*) anIterator->First();
while (nextLibrary != NIL)
{
   // Do something - could retrieve the value associated with this,
   // etc.
   // The "nextLibrary" text object belongs to you
   delete nextLibrary;
   nextLibrary = (TToken*) anIterator->Next();
}

// Request notification whenever a change is made to the set of
// shared libraries
// Assume you have an MMessageTask ready to receive this notification.
TMessageTaskNotification notification(aMsgTask, "SharedLibrary");
environment->NotifyOn(notification);
```

Classes

TEnvironment

`TEnvironment` provides an environment variable registry and notification service. The `TEnvironment` provides methods for adding, updating, removing, and querying of environment variables. Changes to the set of environment variables are persistent across sessions. Applications can also request to be notified whenever any kind of change is made to an individual environment variable (e.g. `"User Name"`) or a category of environment variables (e.g. `Fonts`). Notification is on a per client basis and is persistent for the life time of the client (which is typically the lifetime of an application).

Applications will typically use a `TEnvironmentStack` which provides a shadowing mechanism as well as the functionality of a `TEnvironment`. All values returned by TEnvironment are considered "newed" objects and should be managed by the caller.

```
class TEnvironment : public MCollectible {
   public:
      TEnvironment(char* fileName1);
      TEnvironment();
      virtual  ~TEnvironment();
      virtual  Boolean        Add(const TToken& category,
                                      const TToken& instance,
                                      const MCollectible* value,
                                      Boolean replace = TRUE,
                                      const TToken* replaceDetails);
      virtual  Boolean        Add(const TToken& category,
```

1. Of course, we will be using whatever class the file system provides for representing a file.

```
                                        const MCollectible* value,
                                        Boolean replace = TRUE,
                                        const TToken* replaceDetails);
    virtual  void            Remove(const TToken& category,
                                        const TToken& instance);
    virtual  void            Remove(const TToken& category);
    virtual  MCollectible*   Retrieve(const TToken& category) const;
    virtual  MCollectible*   Retrieve(const TToken& category,
                                        const TToken& instance) const;
    virtual  Boolean         Member(const TToken& category) const;
    virtual  Boolean         Member(const TToken& category,
                                        const TToken& instance) const;
    virtual  Boolean         Members(const TToken& category,
                                        TCollection& result) const;
    virtual  void            NotifyOn(const TNotificationSpecification&);
    virtual  void            NotifyOff(const TNotificationSpecification&);
    virtual  TIterator*      CreateIterator() const;
    virtual  TIterator*      CreateIterator(const TToken& category) const;
};
```

TEnvironmentStack

A `TEnvironmentStack` is a stack of `TEnvironments`. Accessing information in the `TEnvironmentStack` involves trying to find the information in whatever `TEnvironment` is closest to the top of the stack. For example, if you are trying to access the environment variable "`User Name`," the top most `TEnvironment` would be searched, then the next environment on the stack, and so on until the environment variable is found or all `TEnvironments` have been scanned. The `TEnvironmentStack` exports the interface for iterating over all the environment variables in a particular category, notification when an environment variable changes, and synchronous access to the value of an environment variable.

```
class TEnvironmentStack : public MCollectible {
  public:
    TEnvironmentStack();
    virtual  ~TEnvironmentStack();
    virtual  TIterator*      CreateIterator() const;
    virtual  TIterator*      CreateIterator(const TToken& category) const;
    virtual  MCollectible*   Retrieve(const TToken& category) const;
    virtual  MCollectible*   Retrieve(const TToken& category,
                                        const TToken& instance) const;
    virtual  Boolean         Member(const TToken& category) const;
    virtual  Boolean         Member(const TToken& category,
                                        const TToken& instance) const;
    virtual  void            NotifyOn(const TNotificationSpecification&);
    virtual  void            NotifyOff(const TNotificationSpecification&);

    virtual  void                    Push(TEnvironment*);
    virtual  const TEnvironment*     Pop();
    virtual  const TEnvironment*     Peek();
    virtual  TIterator*              CreateEnvironmentIterator() const;
    virtual  void                    StartHere(const TEnvironment*) const;
};
```

TNotificationSpecification

`TNotificationSpecification` is an abstract class that contains the protocol for describing when and how client receive notification when environment variables change. `TNotificationSpecification` subclasses will override the `Notify` methods to do the actual notification dispatching. For example, a subclass of `TNotificationSpecification` could supply notification using Opus IPC messages, post events to an event receiver task or send a packet on the network.

The `TNotificationSpecification` encapsulates information which describes when notification will occur. You can request notification based on a particular category (all changes involving the category `FONTS`); a particular category and operation (all additions to the category `FONTS`); a particular category and instance (all changes involving `FONT Helvetica13`); or a particular category, instance and operation (whenever `SHARED LIBRARY GoodStuff` is updated).

Synchronous notification is not currently supported; however, with enough examples requiring its use, this will be considered.

```
class TNotificationSpecification : public MCollectible {
   public:
      enum      NotificationKind      {kAdd, kRemove, kAddOrRemove,
                                        kUpdate, kAll };
      virtual   ~TNotificationSpecification();
      virtual   void                  Notify(const TToken& category,
                                             NotificationKind kind) = 0;
      virtual   void                  Notify(const TToken& category,
                                             const TToken& instance,
                                             NotificationKind kind) = 0;
      virtual   const TToken&         GetCategory();
      virtual   const TToken*         GetInstance();
      virtual   NotificationKind      GetNotificationKind();

   protected:
      TNotificationSpecification(const TToken& category,
                                 NotificationKind = kAll);
      TNotificationSpecification(const TToken& category,
                                 const TToken& instance,
                                 NotificationKind = kAll);
};
```

TMessageTaskNotification

`TMessageTaskNotification` is a notification specification which will perform notification by using an `TNotificationMessage` (a kind of `MKernelMessage`) send to an `MMessageTask`. The `TNotificationMessage` can be queried to find out what category (and instance) changed as well as the nature of the change (update, addition or removal).

```
class TMessageTaskNotification : public TNotificationSpecification {
   public:
      TMessageTaskNotification(const MMessageTask& who,
                               const TToken& category,
                               const TToken& instance,
```

```
                                      NotificationKind = kAll);
        TMessageTaskNotification(const MMessageTask& who,
                                      const TToken& category,
                                      const TToken& instance,
                                      NotificationKind = kAll);
        virtual  TMessageTaskNotification();
        override void                 Notify(const TToken& category,
                                             NotificationKind kind);
        override void                 Notify(const TToken& category,
                                             const TToken& instance,
                                             NotificationKind kind);
};
```

TNotificationMessage

TNotificationMessage is the kind of message sent to an MMessageTask during notification. It encapsulates the category, instance and notification kind.

```
class TNotificationMessage : public MKernelMessage {
   public:
        TNotificationMessage(const TToken& category, const TToken& instance,
                                    NotificationKind = kAll,
                                    const TToken* replaceDetails = NIL);
        TNotificationMessage(const TToken& category, NotificationKind = kAll,
                                    const TToken* replaceDetails = NIL);
        virtual  ~TNotificationMessage();
        virtual  const TToken&        GetCategory();
        virtual  const TToken*        GetInstance();
        virtual  NotificationKind     GetNotificationKind();
        virtual  const TToken*        GetReplaceDetails();
};
```

56

Babel

Pink International System

by

M. Davis
L. Collins
R. Sonnenschein

56

Contents

Introduction 1
Configuration 1
- Localization Objects 1
- Human Interface 2
- Presentation Languages 2
- TLocale 3

Input 4
- Keyboards 4
  - Creating and Modifying a Keyboard Mapping 6
  - Using a Keyboard Mapping 6
  - Viewing/Editing A Keyboard 7
- Keyboard Transliteraters 7
- Input Methods 8
  - Background 8
  - Interface 10

User Font Editing 13
Text Analysis 13
- Pattern Substitution 13
- Text Service Management 14
- Transliteration 14
- Character Transcoding 15
- Character Properties 16
- Word Boundaries 16
- Text Collation 17
  - Creating a Collation 17
  - Using a TCollater 18
  - Multiple Languages 18
  - Searching 18

Units 18
- Time 19
  - Civil Dates 19
  - Planning Year 19
  - Time Zones 20
  - Calendar 20
  - DateTimeFormat 21
- Numbers 22

Unicode 24
- Background 24
- Alternative Standards 25
- Methods & Status 25
- Design 26
- The Unicode Repertoire 26
  - Future Expansion and Character Registration 28
- Code Assignment 30
- Details 32
  - Paragraph/Line Separators 32
  - Specific Characters 32
  - Ordering of Character Sequences 32
- Sample Code Pages 33

56

Introduction

Pink will be the *first* operating system that fundamentally integrates features necessary for support of native languages worldwide. It starts with the character encoding, which supports scripts and symbols for all languages and typesetting requirements, and extends throughout the toolbox, even up to the development environment (which is traditionally only a 7-bit environment).

Pink provides high-quality layout of text in any language; layout which satisfies the large majority of typographical requirements and can be extended to meet others. Pink also includes other crucial language-sensitive features: text comparison and searching, keyboard mappings, inline input (integrating ideographic text input), etc. It also provides country- or region-specific features such as time and number conversions, and calendar, time zone and daylight-savings support.

This document describes many of the pieces of the international support that have traditionally been referred to as the international utilities (or native-language support or localization support). The other main area of international support is in line layout, which is in a separate chapter. This document should not be the organization of our eventual developer documentation, which should have the goal of integrating each of the international features within the rest of the documentation, to show a fundamental integration within the Pink system.

Configuration

The Pink international utilities contain a number of objects that provide for features of the system (other than text rendering) that are internationally sensitive. Among other facilities, this includes: text comparison; date, time and number formatting; calendars & time zones; keyboard mappings; spelling checkers; preferred fonts; etc.

Most of these localizable objects are table-driven, and rely on persistent data to perform their functions. These objects are a subset of the user-visible objects on the system, which includes documents, other user configurations, etc. Users or localizers may wish to select, copy, edit, install or remove the localization objects.[1] All of the table-driven objects will provide a functional interface for editing the stored information, so that programmers need not know the format details.

Localization Objects

For the purposes of localization, there are a number of relationships which can be organized into a hierarchy. A *script* is a writing system consisting of a number of symbols and conventions for associating them. Examples are Roman (Latin), Cyrillic, Greek, etc.

Each script can be used to represent a number of natural *languages* (for localization purposes, wherever one natural language is written in two scripts we will refer to them as two different languages: one example is Serbian and Croatian, which are essentially the same language, but one is written with Cyrillic and the other with the Roman script). In some cases a script is only used by one language (Japanese), while in others it is used by scores (English, French, German,...).

Each language has a number of *regions* in which the language is used, regions which have different localizations (these correspond to country-codes on the Mac). Examples are the U.S.

---

1 Whenever installed objects are selected, changed, added or deleted, the toolbox will need to synchronize applications and servers.

and U.K., or French, French Canadian, French Belgian, French Swiss, etc. Generally our products are localized to a particular region when we ship them: sometimes applications can be localized to just a language, however, if the differences among the regions of a language are not present in the application.

A *locale* is a part of a region that shares common time and date characteristics: For example, California and Oregon are part of the same zone, while Arizona and New Mexico are not (even though they are in the same time zone, Arizona does not have the same daylight savings as New Mexico). Users will generally personalize their systems by configuring their system differently from the shipping system for their region.

Human Interface

The exact ways in which the human interface is expressed and the system configuration methods implemented is not yet well defined. Most of the international interface elements are not simply the result of isolated international programming efforts, but rather require participation and integration by all Pink system software engineers. Even though the interface and configuration issues are not firm, we can indicate a minimal set of functionality that needs to be present for world-wide markets.

The localization objects will generally be visible and manipulable by users. In some cases users will be able to create and edit objects (date & time formats, for example), while in others they will only be able to choose among different objects (e.g. Gregorian vs. Hebrew calendar). Of the localization objects that are not generally modifiable by users, most will still be table-driven and modifiable by localizers (keyboards, text comparisons, etc.), while some will require coding (calendars, input methods).

Each type of object will have at least one icon (e.g. a pictorial representation) and name. They may have more: in the general case the types will have a number of names, one per installed language (see Presentation languages, below). Similarly, each localization object in each type will have a number of names and icons. Localization objects also have an associated target language (there are some exceptions: some objects are not specific to a language, such as calendars or time zones, and are counted as being in every language). The names for objects specific to a language will generally be in the language, and using characters of the script appropriate to the language.

The system can map from names to objects and back, and can iterate through all the installed localization objects. Target languages can also be used to filter lists of choices, so that human interfaces can present choices within a particular language. For example, the user could choose a filter to see just the English keyboard mappings ("U.S.", "U.K", "U.S. Dvorak", et cetera).

Presentation Languages

Application programs are able to handle many different languages in documents simultaneously. In a word processing program, for example, the user can enter French and German words into the same document: with Pink Unicode text, this will be transparent for non-Roman scripts as well.

However, besides the languages that the user has entered into a document, each application uses a particular language in the menus, dialogs and other user interface elements for presentation of information to the user. This is the language that the application is localized for, and which we will call the *presentation language.*

One of the major advantages of the Macintosh was the ability for well-behaved applications to be localized into any number of languages, changing the interface text *without recoding or even access to the source code.* This localization of the presentation language often involves changing the other interface elements: since the length and content of text strings varies dramatically, elements of dialogs and menus often need rearrangement. Programmers also make

use of routines such as ParamText, that allow localizers to arbitrarily change the order and context of items in dynamically composed strings.

On the Blue Macintosh, there is only one presentation language for an application and for the system as a whole (e.g. system messages, the Finder presentation language, desk accessories, control panel modules, etc.) Applications can change their own presentation languages (and a few do), but there is no system support and no uniformity in interface for doing this.

The ability to change presentation languages is important for two reasons. First, allowing applications and the system to have multiple presentation languages allows both us and our developers to ship one product within a range of different countries, just as the same manual can contain several different languages for use in several different countries. Secondly, in many markets there is a requirement (in some cases governmental) for different people having different native languages to be able use the same product. These reasons are especially important in Europe, where the language support issues are crucial.

Users must be able to select the preferred presentation language in a simple fashion (whatever that eventual interface is: control panel, desktop direct manipulation, etc.). Applications should be notified when the user changes the preferred language, and switch their presentation languages if possible. One system/application can be shipped in a number of languages if at installation time, we allow users to choose the presentation language (before any other text has appeared on the screen, so that they are not faced with confusing menus in a language that they do not understand). They can do this by selecting among different initial languages (with each choice being in the respective language). Applications will also have direct access to lists of supported language and region names.

Note that actually the user will be selecting among regions: however, since most applications are not sensitive to regional differences, the application may have just a generic language presentation covering a number of regions. For example, if the user selects the U.K. the application may just switch to a generic English presentation, or it may switch to a fully local version (its text-to-speech could also start deleting 'R's with wild abandon).

The development system will need provide support for multiple languages and localization in general. For example, the International Software Support group has developed a number of things that make the translation process easier. To start with, there is a lexicon of Macintosh terms that should be used for consistency when translating. For example, you don't want "click-and-drag" sometimes translated and sometimes *touch-and-tug*; or "cut-and-paste" translated as *wound and glue*. This kind of lexicon should be easily accessable from within the development environment. For more easily translating successive versions of a product there are MPW scripts that automatically translate identical strings. For example, if string #24 in a 1.0 English version and a 2.0 English version match, then the scripts will automatically translate string #24 in the French 2.0 version by using the 1.0 French version. In addition, the ability to annotate strings with explanations that are not visible to the user, but can be seen by localizers will be extremely helpful.

For multiple languages within the same program, the development environment should also make sure that the translations stay in sync: if I add a string in the English version, then some string needs to be added to the French, and the localizers need to be able to find out what it is.

TLocale

Access to the international objects is through the Rainbow Warrior environments mechanism. By using this, for example, a programmer can iterate through all of the keyboards available on the system, or iterate through all of the number formats. One of the accessable objects is a TLocale. Each TLocale contains a set of international objects that are associated with a particular region (e.g. "Britain", "US", "Schwytz", "Paris", "Texas").

The TLocale mechanism allows users to change between different sets of international configurations with one choice. It also allows a set of objects to be packaged as one entity for

installation: adding Japanese to a system can add one or more Japanese TLocales, including the necessary keyboards, input methods, calendars, fonts, et cetera.

For each language there is a preferred TLocale. This allows programs to determine a choice of items such as spelling checker based on language. For circumstances where no language or TLocale has been chosen, there is always a system default TLocale. A style-run of text that has no associated language is assumed to be in the language of the system TLocale, for example.

Methods are provided for creating a new TLocale, iterating through the objects in a TLocale, changing the preferred TLocale for a language, and for changing the contents of a TLocale (e.g. changing the "American" keyboard mapping from "Querty" to "Dvorak"). The capabilities provided by these methods will be provided to users by a view.

To use each of the international objects, applications will commonly access the system TLocale through Rainbow Warrior. They will then select the particular type of international object that they require: keyboard, collation, number format, calendar, etc.

Input

Input of characters uses keyboards, transliterators and input methods. Keyboards are used for the fundamental conversion of keystrokes to characters; transliterators are used for dead-key support and phonetic conversions; and input methods are used for large character set support (eg. the 10-40 thousand characters used by CJK (Chinese, Japanese & Korean) scripts.

Note that all of these objects are invertable to some extent: it is possible to ask a keyboard, for example, for a sequence of <key, modifier set> combinations that will produce a given string. The keyboard will return a sequence that suffices. In general there may be no sequences, one sequence or many sequences. If there are more than one sequence, then the keyboard will try to pick a "simple" one, with the least complicated modifier sets. If there are no sequences, then the answer will indicate so, and invert as much of the string as possible. This will also be done by transliterators and input methods: asked how to produce a given string, they will return a string containing a set of characters that will generate the target string, if possible.

Keyboards

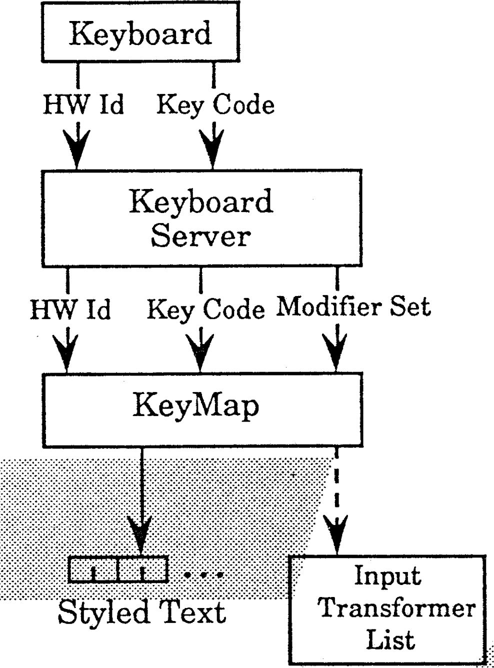
A KeyMap contains the mapping from virtual keys to text.[2]

KeyMaps may also contain a reference to an input transformer list. This input transformer list consists of zero or more transliteraters and an optional keyboard input method used to perform inline input. Users can edit keyboards, removing or adding transliteraters, input methods, or keycode mappings.

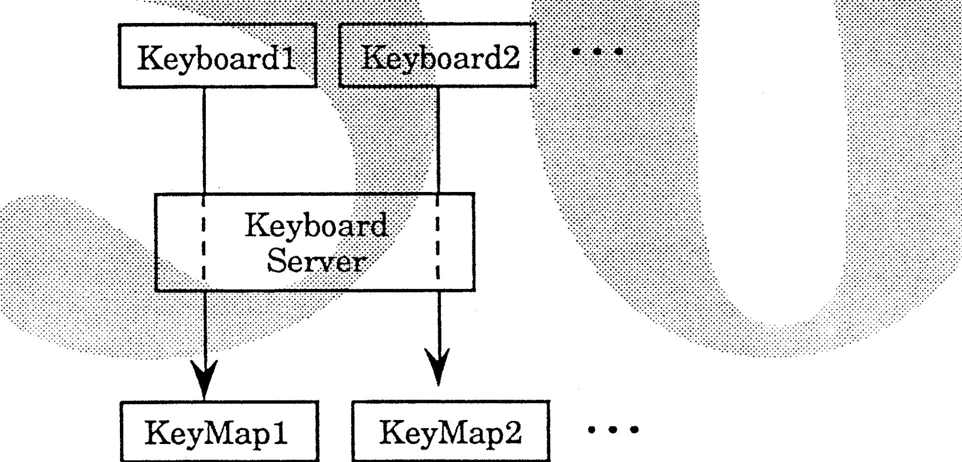
Note: transliteraters will be used to provide the functionality of "dead keys" on the Macintosh.

---

2 There is also a mapping between the hardward codes emitted from the hardware and the so-called virtual keycodes. To see a discussion of this, consult the Toolbox.



There may be multiple keyboard mappings active at once, each associated with a given piece of hardware (one of the keyboards attached to the system). However, these keyboard mappings are common across all applications, unless a specific application overrides the standard mappings (e.g. terminal emulation programs). Users can choose which keyboard mapping to attach to which keyboard. The choices are maintained over reboot.



The system TLocale's keyboard is used by default on the shipping system. Note that from the Mac hardware we can determine the type of keyboards connected (standard, extended, ISO, etc.) but we cannot determine the configuration of letters used on the keycaps (English, French, German), so users have to select the default keyboard mappings manually.[3]

According to the current ToolBox model, the choice of modifier keys is fixed, and limited to the current Mac set of 8 modifiers: shift, option, command, caps-lock, control, right-shift, right-option, right-command: we expect this to change to at least 16 modifiers.

---

3 There is undoubtedly a good reason the hardware folks had for this; we are probably saving hundreds of cents per keyboard by not having different hardward ids for different keycap plastics.

The user interface for choosing keyboards and displaying them is not yet well defined. Two points are clear: there should be visible feedback as to the current keyboard, and there must be a fast "power-key" method of choosing keyboards (in addition to a Control-Panel style interface), since in many cases keyboards are switched very often. The keycaps-equivalent should allow type-through, so that the user can type (or click) characters on the keycaps keyboard and have the characters pass through to the next layer. Keyboards must also be switched synchronously: on the Mac the keyboard switching can lag behind key input, so that keys are mapped incorrectly.

Creating and Modifying a Keyboard Mapping

To create a KeyMap, the programmer specifies a default language. He then adds mappings to a KeyMap by successively specifying the following:

a) a virtual keycode

b) the modifiers that must be on for a match

c) the modifiers that must be off for a match

d) (optionally) a hardware keyboard id

e) a result

The keyboard id is optional, and intended for special mappings for different hardware keyboards (so that we don't have to have a hack like the Mac itlk resource). The result can be either a single character or a string of styled text.

The programmer can also iterate through a KeyMap to recover mappings, although the KeyMap may perform some optimizations, so that the mappings that go in may not be the mappings that come out.

The programmer can also add or remove transliteraters, and add or remove an input method.

Using a Keyboard Mapping

When an application or a keyboard server supplies a virtual key to the KeyMap, it returns one or more characters. Since an item of type TStyledText can be returned, a keyboard can return text of various lengths, fonts, styles, etc. (The Toolbox provides separate mappings from hardware to virtual keys, and will supply separate calls to determine which keys are down.)

There are two other interface changes from the Mac. First, events will be generated with all key downs (on the Mac the application never gets a chance to see a dead key-down, unless it patches out KeyTrans).

Secondly, command keys will be treated differently. On the Mac, command keys are not sensitive to shift, but are sensitive to caps-lock: just the reverse of what should be done. The menu manager also up-shifts and strips diacriticals when matching command-keys, eliminating the possibility of distinguishing between ⌘ a, ⌘ A and ⌘ Ä. We will let application and menu manager easily determine what a key would have been without particular modifiers, so that it can determine that the user hit ⌘⇧⌥ Q (Command-shift-option-Q), without having to know that it happens to be ⌘ Œ on the current keyboard.

Command keys also use a different mapping on the Mac than non-command keys; but this is only so that there are no command-dead-key combinations. This difference is not necessary in Pink. In addition, command keys need to be relatively constant: it would be very disturbing for users to have them change depending on the current keyboard (e.g. if I have a dingbats keyboard, I do

not want to get ⌘ ☎ instead of ⌘ A). To support this, users will be able to select their command keyboard separately from their current keyboard.[4]

The text classes will allow users to associate keyboards with fonts or styles, so that they can be automatically switched if the user desires. That is, when I click down in some Symbol text, I would like to automatically get the symbol keyboard, when I click down in some Russian text, I would like to get a Russian keyboard, etc. We will provide utilities for use by the text classes and applications to find a keyboard that contains a given character so that this can be done.

Viewing/Editing A Keyboard

There will be the equivalent of the keycaps desk accessory to allow users to view the current keyboard. Multiple keycaps can be open at the same time. A check box will allow the keyboard to type through to the next frontmost window.

In addition, an extra command/button will allow the keys to be edited. The editing paradigm will be that a key can be set by pasting, inserting, or dragging text onto the key position. We will ship keyboards that contain all of the (non-ideographic) characters represented by the fonts we ship with the system, so users can drag from one key on one keyboard to set a key on another.

A side panel or desktop object can also be shown that has a list of characters which can be dragged onto keycaps. Since the character set is so large, a number of organizations of this list can be used:

a) Ordered by character code

b) Standard sorting order

c) Ordered by group (script or other grouping: mathematical symbols, etc.), then ordered by (a) or (b).

d) Customizable order.

Keyboard Transliteraters

On the Macintosh, dead keys are used to extend the range of the keyboard for accented characters. With this mechanism, a user can type a key (e.g. option-u for umlaut) which puts the keyboard into a special state, but does not generate a character or any other visible indication of what has occurred. When the user then types a base character—one that combines with the accent—then the keyboard generates the resulting accented character (e.g. option-u, e produces ë).

Dead-Key Example:

Key	Deadkey state	Display
b	<none>	b
option-u	<umlaut>	b
a	<none>	bä
d	<none>	bäd

---

4 Certain command-keys will need to be reserved for the use of keyboard switching, input methods and transliterations. Since these keys are very frequent when entering text, they need to be easily accessable. On the Mac, we use various combinations of ⌘-space with other modifier combinations ⇧, ⌥, and ⌃.

In Pink, this modal mechanism is replaced by the use of transliteraters. When the application inserts characters from a keyboard into some text, then it will call the list of transliteraters associated with that keyboard (and then the input method for the keyboard—see below). An accent transliterater can provide the same functionality as dead keys, in the following way. When the user types an accent, for example, the keyboard will generate a spacing accent,which is entered into the text stream. When the user next types a base character, then it is entered into the text. However, the text classes (or applications that don't use the text classes) will then call the keyboard's transliterater, which will replace the accent and base character by an accented character.

Transliterater Example:

Key	Pre-transliterate	Display
b	b	b
option-u	b¨	b¨
a	b¨a	bä
d	bäd	bäd

Transliteraters can perform many other functions: for example, they can replace generic quotes (".') by right- and lefthand quotes (",',"). They can also be used to perform general script transcriptions for any cases where the transcription is simple and unambiguous, as when converting from romaji to katakana or hiragana for Japanese, converting Hangul (letter components) to Jamo (letter syllables) for Korean, or converting Querty to Hebrew, etc.

Input Methods

A keyboard input method supplies a mechanism for converting from keyboard characters to ideograms: generally the Han ideograms used in Japanese, Chinese and Korean. These ideograms may have originated as drawings of objects associated with a word, but the original association is now not generally recognizable.

Background

For those who are unacquainted with the requirements of Han character input, imagine for a moment that English were written with logographs (aka ideographs): that is, each separate word or perhaps syllable was represented by a separate picture, and is written without spaces. The number of English words is clearly too large to represent on a keyboard, so a number of different methods arise to input characters.

The most popular of these allows the user to type in characters in a phonetic transcription, which is automatically translated into ideograms (other methods—including handwriting analysis—are also possible, and can be used in addition).Let's suppose that we wanted to enter the ideographic sentence:

The artificial logographs in this example are the in the following table. We also include real chinese logographs for comparison.

Fake Logograph	English	Chinese Logograph
1	a/an	一

	neighbor (man/house)	鄰居 (side/dwell)
	read	讀
	she/her	她
	address	地址 (ground/location)

We would type our desired sentence in phonetically as: *unayburredhurudres* (remember that there are no spaces!). As we type, the input method parses the text grammatically, and converts phrases to the corresponding ideograms (as far as it can). Since there may be a number of renderings, the processed text may change as we type in.

Type-In	Screen Display	Reading
u	1	A
un	1	An
una	1a	A·na
unay	1	A·neigh
unayb	1 b	A·neigh-b
unaybu	1 bu	A·neigh-bo
unaybur	1	A·neighbor
unayburr	1 r	A·neighbor-r
unayburre	1 re	A·neighbor-re
unayburred	1	A·neighbor·red
. . .	. . .	. . .
unayburredhur	1	A·neighbor·red·her
. . .	. . .	. . .
unayburredhurudres	1 1	A·neighbor·read·her·a ·dress

Even once the sentence is complete, the ideograms are not correct. It requires additional knowledge beyond syntax to get a correct result, since syntactically both the forms:

Det.·Noun·Verb·In. Object· Det.·Noun e.g. "A neighbor read her a book", and

Det.·Noun·Verb·Pos. Pronoun· Noun e.g. "A neighbor read her address"

are syntactically correct. So, the user needs to be able to modify the transscription by choosing alternative renderings.

With current technology, this is done by either modifying the phrase or segment length (in this case, indicating that *udres* is one word), or by selecting an alternative ideogram (e.g. manually choosing [ ] instead of ( )).

Unfortunately, most sentences are much more complex than this—especially in Japanese—and have many more alternative phrase lengths and alternative readings. The number of homophones (ideograms with the same pronunciation) in Japanese and Chinese are very large: up to one hundred readings for the same symbol! Even when alternatives are chosen, the input method uses its knowledge of the grammatical structure of the sentence to filter out inappropriate options; otherwise the user would be swamped. (The user can choose to see all options for unusual readings.)

In addition, many Japanese users are more comfortable with the QWERTY Roman layout than with the native phonetic keyboard layout for Japanese. So, transliteraters allow users to type in with Roman characters *(romaji)*, which are transliterated to Japanese phonetic characters (*kana*).

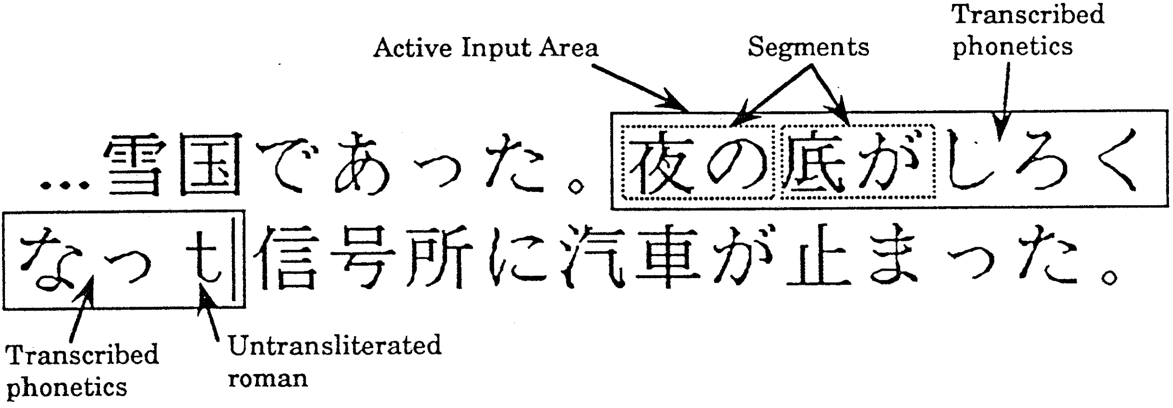
Interface

An input method can be used by the text classes to provide inline input, whereby the text is entered and converted directly in the document (as opposed to in a separate popup window). The keyboard characters generally represent the phonetic structure of a sentence to be parsed and replaced by ideograms; they may also represent the visual appearance of the ideograms, as in stroke input methods. Alternative methods of entry are also accomodated: some methods have the user draw the characters on a tablet, for example. In those cases, the characters are already correct before the text classes encounter them. (We may need to offer a "floating window" for inline conversion for those applications that do not use the standard text classes and do not implement inline input correctly: it depends on how strong we can be about eliminating such applications.)

The application or high-level text classes are responsible for implementing the user interface for input methods. This will necessitate the ability to handle several different types of selections, and different kinds of highlighting to distinguish them. At any given time, there are a number of different kinds of text on the screen. There are:

- inactive text (that is outside the text targeted for conversion)
- active converted phrases (for which alternatives or different lengths can be chosen)
- unconverted text (generally phonetic),
- untransliterated text (generally Roman).

The figure shows how Japanese inline input might look while in progress. The sentence being input, called the *active input area*, is outlined with a solid line. Notice that it starts in the middle of the first line and ends in the middle of the second line. The text outside of the active input area does not participate in the conversion process. The phrases which have already been converted, and the un-converted input are outlined with dotted lines. The un-converted phonetic input is followed by a single *romaji* letter. The insertion point is after the *romaji* letter.



The following describes how a user would use such a system to type Japanese a sentence at a time. This is only one possible scenario for how the user input would work.

As the user types text it is displayed in the application window. The active input area should be visually distinct from the rest of the application text so that the user can see which text is being processed. As text is processed, the contents of the active input area may change; for example, as romaji input is transcribed into kana by transliteraters, or as new segments are identified and converted by the input method.

When the input method generates segments, it may not generate the correct segment boundaries, or it may not choose the correct homophone for a given segment. In these cases the user must be able to make adjustments. First, the user must indicate which segment is to be adjusted. This could be done by pointing at the segment with the mouse, or by typing some sequence of commands, such as the left and right arrow keys. However the segment is indicated by the user, it must be given a visual appearance different from the rest of the document text, and the rest of the text in the active input area.

The user can change the segmentation by indicating that the indicated segment should be made one (phonetic) character longer or shorter. (Note that this makes the following segment one character shorter or longer, and is thus the same as moving the segment boundary.) This could be done by dragging the segment boundary with the mouse, or by typing some sequence of commands, such as the up and down arrow keys. Depending upon the particular input method, changing the length of a segment may cause all the segments following it to change as well.

The user can display alternate homophones for a segment by double-clicking on the segment, or by typing some sequence of commands, like the enter key. Alternatives can be selected either by changing to the next homophone in-place, or by bringing up a floating window displaying all homophones, from which the user would then choose the correct homophone.

Text editing within the active input area cannot be the same as text editing elsewhere in the application window because of the special actions the user needs to perform on the active text. The distinctive visual appearance of the active input area indicates to the user that this is the case.

Different levels of functionality could be supported by different input methods. For example if the user were to delete a character from the middle of a segment, the information that an input method maintains to process the active input area could become invalid. On the other hand, most input methods will probably allow the user to freely edit the un-converted input. Less sophisticated input methods might wait until the whole sentence is typed before doing any segmentation, while other input methods might generate segments "on the fly". Some input methods might decide to "drop" segments off the front of the active input area as the user types in order to limit the amount of internal storage needed.

The user can request that the un-converted input be converted by some keyboard command. It probably makes sense to have the command be a modification of the command that requests alternate homophones for a segment.

The user can indicate that conversion is complete by some command like return or enter. When this is done the distinctive visual appearance of the active input area and the active segment would be removed and the caret would be placed after the newly converted text.[5]

In summary, users need to be able to perform the following actions on the active input area:

- Type in
- Change the transcription method or the input method
- Change the length of a segment
- Choose another homophone for a segment
- Edit text in active input area
- Convert un-converted input
- Indicate that conversion is complete

Applications need to communicate these actions to the input method, and we need to provide a standard human interface for input methods that does not conflict with application interfaces. Methods for performing these actions include pointing and clicking with the mouse, and typing cursor keys and command keys. Keyboard-driven commands are extremely important for input methods: a user should not have to take his or her hands from the keyboard in order to enter text! Note that many of these actions may have different meanings outside the context of the active input area, although we will try to minimize the modality.

Users must also be able to switch between different input methods and transliteraters fairly quickly, since it is often the case that users do not know the phonetic transcription for a particular character, for example, and need to switch to an alternative method just for that character. Input methods also provide dictionary management facilities, allowing users to edit the dictionaries used by the method. For example, users will often enter in new phrases into dictionaries: a user may indicate a phonetic key, the result phrase, and the grammatical category, for example. They may also want to delete phrases, and select or deselect different dictionaries. Dictionaries of specialized terms (medical, legal, etc.) can also be bought independent of the input method.

In the case of inline input, the application and the input service must cooperate closely. Only the input service changes the active input area and the segment boundaries (except for complete deactivation); only the application knows where characters are on the screen and how to alter their visual appearance: when the user uses the mouse to indicate a particular segment, only the application can do the mapping between mouse position and character positions.

Given this division of knowledge, the input service must be able to request the following functions from the application:

---

5 To contrast this with the current situation in Blue, the majority of applications there depend upon a floating conversion window which pops up with any keystroke. This window supports the same range of features that we discussed above, but the lack of integration causes a number of human interface problems. It would be like having to bring up a dialog every time you wanted to enter some text into a document: it is very modal and quite distracting.

A number of applications on the Mac and on other machines have already incorporated inline input methods.

- Replace the text in the active input area or in segments
- Change the segmentation in the active input area
- Change the active input area (e.g. change active text to inactive)

The application needs to keep the appearance of the screen consonant with the text and segmentation structure required by the input method, including special highlighting for visual feedback of this structure.

User Font Editing

Unfortunately, the set of ideographs in use on Asian systems is not a closed set. For example, obscure variants of characters are used for personal names; and, unaccountably, people wish to be able to type in their own names (as many people in Pink have graciously pointed out, this would all be much easier if they just used English).

In order to do this, end-users must be able to make up new characters. We have this capability on the Mac, and must extend it to Pink. Luckily, editing these characters need not call on the same kind of sophisticated design talents that are needed for outline font creation. These characters are formed out of standard components, (called *radicals*), so users can select, position and scale the pre-formed radicals from the font to form a new character in that font.

Once this has been done, and the user has picked a Unicode character code, then the character can be entered in keyboards and/or input method dictionary. The user can either choose to pick a character code randomly from the Unicode user range, or pick a specific character code. A specific character code can be chosen in case the character code is assigned in Unicode and the user just does not have that character in his font, or can be assigned out of user-space.

In order to avoid collisions, it would be preferable to store graphic information with the document or send it in transmission. For example, if a document can store a set of user-definable unicodes and their graphic representation and a unique stamp, then in case of collision with characters defined on other systems the graphic can be used.

Text Analysis

Text analysis classes provide for pattern substitution, transliteration, character transcoding, character properties, word boundaries and text collation.

Pattern Substitution

Applications often wish to present composed messages to the user, where a number of strings are concatenated. However, simple concatenation neglects the fact that different elements of a sentence are presented in different orders in different languages. To avoid this, the application should use substitution within a string, where the order and context can be localized to fit a particular language.[6] For example, a filename and disk names would be substituted within the text *"Copying file from* ***sourceDisk*** *to* ***targetDisk****"*. In another language, the order of the element of this sentence might be: *"Onto* ***targetDisk*** *from* ***sourceDisk*** ***file*** *to-copy"*. Pattern substitution can cover most cases where messages must be composed from dynamic elements.[7]

---

6 On the Blue Mac, this roughly corresponds to ParamText, which is limited to use with dialogs and unstyled text.

7 Although pattern substitution goes a long way towards satisfying the requirements of composing messages in different languages, a full solution would take the grammar of the

Since use of styled text will be universal in the interface, we propose using special styles (or even icons?) to represent the replaced elements, instead of special symbols (^0, ^1,..., ^9 are used on the Mac). That is, rather than type in special characters, in order to form a pattern the user would select the text and mark it as variable (meaning that it will be replaced in pattern substitution.

If there is this special style, then a pattern could be preprocessed to determine the locations of the elements to be replaced. The contents of the style could indicate formatting options for numbers and other generated elements. The style would have to be non-merging: two adjacent runs having that style should not merge internally.

Text Service Management

As far as we understand from the System Architect, text services such as spelling checking, hyphenation, grammar checking, translation aids, lexicons, root extraction and noise word filtering (for context retrieval), title filtering (for title text—see below), etc. will be handled automatically by the toolbox. These services are, of course, very language sensitive, so we assume that the international group will be working closely together with the toolbox group to define the interfaces.

Transliteration

Transliteration is used to convert between scripts that can be used to represent a given language. This is especially important for those languages such as Japanese that use a Roman keyboard to key in text, which is then transcribed into native Japanese characters. These characters can also be converted back into Roman characters. A particular class of transliterations called input transliterations obey two requirements.

- Uniqueness
  Transcription from native to foreign script is unambiguous. Two different native strings cannot be transcribed to the same foreign string. For example, if the native script distinguishes between a retroflex and a dental T, then a transliteration cannot map them onto the same symbol "t".
- Completeness
  Transcription from native to foreign script, or from foreign to native is complete. Every sequence of native symbols will map onto some string of foreign symbols. Transcription from foreign to native script should be complete, but is not generally unambiguous. For example, when a Roman-to-Japanese transcription is used, "ra" and "la" map onto the same Japanese symbol.

A TTransliterater object is used to perform transliterations. The transliteration is be composed of a set of context-sensitive rules, which are designed so that knowledgeable non-programmers can edit them reasonably for localization.

Examples of rules:

cho ➡ ちょ

---

sentence into account, substituting different text when the replaced elements have different number, gender, etc.

The forms of agreement are not easy to predict for your average programmer: for example, some languages have three different numbers: singular, dual and plural. Even English originally had this: instead of just two first-person pronouns *me* and *us*, there were *mé*, *unc*, and *ús*, where *unc* meant *the two of us*. We will not attempt a "full solution" in Pink 1.0.

t[t] ➟ っ

to ➟ と

Using these rules, chotto can be transliterated into ちょっと.

Transliteration may be dependent not only on script but also on language. It is also inherently an *n* x *n* problem: expecting to transliterate from Russian to Hindi by using a series of Cyrillic-Roman and Roman-Hindi transliterations is doomed to failure, since the transcriptions used to represent non-Roman letters will vary depending on the script being represented: in some cases *th* will represent the sound found in *thick*, while in others it is an asperated *t*.

Transliteration can be used not only for inter-script transliterations, but also for intra-script transformations, such as for accents (replacing dead-key behavior—see Keyboards), or to change generic quotes ('book') to directed quotes ('book'), or even glossary-style behavior.[8]

They are also used for language-specific processing such as converting text from upper to lower case and back, creating title text (E.g. "With the Important Words Capitalized"[9]), and stripping diacritical marks. Note that when these facilities are only desired for display, then these transliteraters could be applied to the display form and not the backing-store, maintaining the original form of the text internally.

Character Transcoding

A TTranscoding converts from Unicodes to a foreign character encoding (such as Blue) and back. Note that individual character codes should not be converted in isolation: sometimes one UniChar converts to multiple foreign character codes, or vice versa. Each TTranscoding consists of a number of CodeSets, where the latter converts a restricted range of characters with a consistent format. For example, for conversion to Blue, one code set converts to ASCII (Roman) while another converts to JIS (Japanese), while another converts to Symbol. Note that an exact conversion round-trip cannot be guaranteed: although Unicode is a superset of other character encoding standards, there are multiple representations of the same text in different standards. The round-trip from a foreign standard into Unicode and back should result in an essentially equivalent string, however.

The conversions are low-level conversion methods: they do not translate text styles, with the exception of certain font-based information. The general conversions for styled text, as when converting the clipboard between Pink and Blue, is up to higher level text classes.

The Blue character encoding and others do use fonts to distinguish code set. When converting from foreign encodings, an optional list of font numbers and text starts can be supplied. The code conversion can use this information to code sets for characters such as "Γ" in the Symbol font. When UniChars are converted to a foreign encoding, a similar list of font numbers and text starts can be generated.

When the foreign character encoding uses in-line character set shifts (such as ISO 8859, which uses escape sequences to shift), then the font specifications are neither used nor produced.

Note: the main interface differences between Transcoding and Transliterating are that transcoding converts to an arbitrary byte-stream instead of another TBaseText object.

---

8 Whereby abbreviations can be automatically expanded as you type. For example, whenever I type "SANE." a transliterater could convert that text to "Standard Apple Numerics Environment."

9 True title text requires use of a title-filter text service in order to eliminate particles. Otherwise the titled text will appear as: "With The Important Words Capitalized."

Character Properties

Additionally, a number of global character properties are accessable, indicating which classes (in the non-OOP sense) characters belong to. These properties are independent of language, but might be installed with particular languages. That is, the properties for an Ethiopian letter might not be available unless the Ethiopian language is installed. The properties include such items as:

1. classifications: letters, diacritical marks, digits, whitespace, etc.
2. directions: character direction, run direction (for use in layout)

<It is an open issue where there ought to be a single table for the system, or that there is a collection of modular tables, where the installation of a language may add a new table.>

Word Boundaries

Even in English, word boundaries are ambiguous: without dictionary word-lookup and grammatical parsing it is unclear whether a generic period ends a sentence or is part of an appreviation word ("etc.", "U.S.A.").[10] However, the word boundaries facilities are subject to a certain constraints: they must be very fast in the general case (even at the expense of minor anomalities), and they must be easily modifiable for localization. The standard implementation uses a state-table mechanism that allows either the class-mapping or the state table to be easily modified for different languages or special purposes. For other languages or purposes where word-break is much more complex (e.g. requiring a dictionary lookup), the standard mechanism may be overridden.

Internally, the standard word state table takes <Unicode, Language> pairs onto a standard set of word classes. It then maps from a <state, word-class> pair onto a new state and an action. Two different start states are used depending on whether we are going forwards or backwards. Clients can simply use the standard table, construct their own, or completely override the classes.

The WordBreak object is used to determine the boundaries between words. Different objects can be used either for word selection or for word-wrapping. The standard methods use a state-table mechanism for speed, and allow distinctions of words such as the following (where the vertical bar indicates a break):

Source: A U.S.A. $55,000.00 re-education is (clearly) enough,
Select: |A| |U.S.A|.| |$55,000.00| |re-education| |is| |(|clearly|)| |enough|,| |
Wrap: |A |U.S.A. |$55,000.00 |re-|education |is |(clearly) |enough, |

To use the standard word iterator, you will use ask TLocale for a wordmap object. Once you have it, you will create a word iterator from it. You will then provide the text and offset that you are starting from, and whether to use the previous or following word if that offset is on a boundary. You can then call Current to the get the word break around that offset, or Next, Previous, and Nth to get successive word breaks. Nth(0) is equivalent to Current; Nth(1) is equivalent to Next, and Nth(-1) is equivalent to Previous. First will reset the current word to the first word in the text, while Last will reset it to the last. So, to get the 3rd word, call First, then Nth(3); for the third from the end, call Last, then Nth(-3).[11]

---

10 Unicode does have specific periods as well as the generic period: if an abbreviation period is used, then there is no ambiguity.

11 We could add utility routines to do NthFromStart and NthBeforeEnd if people thought them useful.

In addition, calls will be provided for doing language-sensitive intelligent cut & paste. Intellegent cut & paste ensures that cutting a word preserves word-breaks and minimizes spaces, while pasting will insure word breaks around a pasted word. For Roman and many other scripts this involves inserting or deleting spaces where needed: other scripts may not use spaces the same way.

Open issue: we could support sentence selection using essentially the same mechanisms. In general sentences are not as clearly distinguishable in some languages (such as English, etc.) because of ambiguities between sentence period and abbreviation period, but it would be more international than what MS Word does now, for example.

Text Collation

Text collation ordering includes provision for a number of very significant features required for proper comparison and sorting of text.

a) strong ordering (Ax << Bx)

b) weak ordering (Ax < ax < áx << Axx),

c) grouped characters (cx << chx << d),

d) expanding characters (aex < æx << aexx),

e) ignored characters (ax < a-x < a-xy).

Expanding characters are treated as if they are inserted after the new string when comparing. We will use the notation "æ"/"e" to mean that "æ" is compared by comparing the "æ" character and inserting the "e" afterwards. In the case where "a" < "æ"/"e", the text "Bæk" is ordered as if it were convered to "Bæek", with the "æ" ordered weakly after "a".

The specific characters that have these behaviors are dependent on the specific language: "a" < "ä" is a weak ordering in German, but not in Swedish; "ch" is a grouped character in Spanish, but not in English, etc. Orderings can also differ within a language: users may want a modified German ordering, for example, to get the older standard where "ä" is treated as an expanding character (aex < äx < aexx).

Note that no character encoding contains enough information to provide good alphabetical ordering: for example in the Macintosh, simple bytewise comparison yields:

a) "A" < "Z" < "a" < "z" < "Ñ" < "Ø" < "Å".

The standard collater does not include the capability for dictionary-based collation, which may be required when the collation order is not deducible from the characters in the text. For example, the abbreviation *St.* is ambiguous, and may be sorted either as Saint, St. or Street. This behavior, however, could be subclassed.

Creating a Collation

The programmer can append a new string to a TCollater, in which case it is ordered after the very last string. The programmer must specify if it is a weak ordering ("a" < "A"), and whether the string has expanding characters ("a" < "æ"/"e"). Null ($0000) is a special character, which is inserted in a TCollation at creation time. It is always ignorable, and any character only weakly greater than it will also be ignorable. Any character not in the TCollater will be ordered strongly before all other character codes.

For example, we could build a new TCollater as follows:

Add		Result
< a	=>	null << a
< ä	=>	null << a < ä
< æ/e	=>	null << a < ä < æ/e

Rule		Result
< A	=>	null << a < ä < æ/e < A
<< b	=>	null << a < ä < æ/e < A << b
< B	=>	null << a < ä < æ/e < A << b < B
<< c	=>	null << a < ä < æ/e < A << b < B << c
< C	=>	null << a < ä < æ/e < A << b < B << c < C
<< ch	=>	null << a < ä < æ/e < A << b < B << c < C << ch
< cH	=>	null << a < ä < æ/e < A << b < B << c < C << ch < cH
< CH	=>	null << a < ä < æ/e < A << b < B << c < C << ch < cH < CH

Using a TCollater

To use the standard collation, you will ask the environments mechanism for your TLocale, then ask the TLocale for a TCollater. In simple comparison, the TCollater takes two strings and compares them. If the user supplies the two optional text indices, then the method will return the positions in the two strings where the crucial difference is found. The crucial difference is the first point at which there is a strong difference, if there is one. Otherwise it is the first point at which there is a weak difference, if there is one. Otherwise, the strings are bitwise identical (and if supplied, the optional text indices will return the lengths of the two pieces of text).

Multiple Languages

A TLanguageCollater can be used to compare styled text that has runs of different languages. Each TLanguageCollater contains an ordered sequence of TCollaters, each associated with exactly one language. When comparing styled text, the following conventions are used:

- A character with no language (or with a language with no TCollater) is ordered using the first language in the TLanguageCollater.
- Two characters of the same language use the associated Collater.
- Two characters of different languages use the ranking of the two TCollaters within the TLanguageCollater , unless the characters are identical and each is not a member of a grouped character. In the latter case they are treated as equal. (e.g. Spanish space and English space are equal).

Searching

Clients could also use a TCollater or TLanguageCollater for searching, but it is not optimized for that. We will offer routines for faster searching, but this will probably not extend to inclusion of features such as wildcards or regular expressions. Unfortunately, a simpleminded application of faster algorithms such as Boyer-Moore can preclude certain language features such as grouped, expanding and ignorable characters, so we are still investigating to see how these algorithms can be modified to account for these features. Whenever those language features are not in use, then we will use a 'fast path'.

Units

A Unit can associate unit quantities and their textual representation in a given language, and perform conversions to other units. Both plain and abbreviated forms can be included. For example, meter can map to "meter", "metre" or the Greek, Cyrillic, etc. equivalent; USDollar can map to "$" or "$US", etc. A TLocale can also specify the default units for a given measure: English vs. Metric, the default currency, etc.

In terms of conversions, units generally fall into three categories: simple units such as feet or meters, predictable units such as times or dates[12], and volatile units such as currency rates or Apple stock prices. We will initially support time and date conversions, and perhaps some degree of simple unit conversions: volatile conversions are not exactly high on our priority list.

Time

Time and date are special units, and require more support. The basic system time is measured in seconds since an arbitrary base date (e.g. 00:00:00 Jan 1, 1904 on the Mac[13]) and require a dynamic range of approximately 10^{13} seconds before and after the present (a comp—64-bit two's-complement integer—is used on the Mac). See the Toolbox time routines for more information.

Time zones, Calendars and DateTimeFormats are used to convert this form into specific human measurements (see below).

Civil Dates

Civil dates are used to determine a particular day of the year according to a given calendar. They can be used to determine a variety of dates that are algorithmically derived from calendar values, such as holidays (Thanksgiving, Christmas, etc) for project planning, start and stop dates for daylight-savings time.

Note that in some cases a civil date may require a calendar which is not standard for the given local: for example, even in a country that uses the Gregorian calendar, Easter or Hebrew holidays will depend upon different calendars.

Civil dates are based upon simple rules for determining a date in a given year. To constuct a civil date, the client will add rules that specify dates according to a calendar during given spans of years. Once constructed, a civil date can be iterated from a given TTime, either forward or backward. For example, given a TTime corresponding to Jan 1, 1983 and a civil date "Thanksgiving", the client can find the first instance of the civil date after the TTime.

The system will be shipped with a set of localized civil dates for given countries. Users can also add new civil dates to the system. (It is easy to supply degenerate rules which simply list a set of specific dates in given years, or to give slightly more sophisticated rules like: the last Thursday in November.)

Planning Year

A planning year is a collection of civil dates and work-week information that can be used for project planning. Planning years can be created or edited by users and applications. Planning years will return the number of workdays between any two TTimes, and also allow iteration through workdays.

---

12 Strictly speaking this is not true, since with some lunar calendars still in use, the date depends on the *visible* phases of the moon. With bad weather, the month in a particular location can start as much as three days later. We do handle this as a special case.

13 The choice of origin is not particularly important, and will be hidden by the Toolbox classes.

Time Zones

A TDaylightZone maps from a GMT[14] datetime to a local datetime and back. TDaylightZones are of finer granularity than regular time zones, since any two locations are in the same zone just in case they currently have the same daylight-savings periods, and have had the same in the past. For example, Arizona and Colorado have different zones, even though they are in the same time zone, because Arizona does not have daylight-savings time. Zones internally use a collection of civil dates to find the starting and ending dates for daylight savings time. This functionality provided by zones is a superset of the UNIX time zone mechanism, and is also table-driven.

In addition, zones also contain geographic information which allows us to provide a direct-manipulation interface for choosing zones. The human interface for setting the clock time should encourage users to set their location, since otherwise date stamps on files and network transmissions will be incorrect. We will ship a number of zones to satisfy most needs; it is unclear whether we can make the human interface easy enough to allow user-creation. As opposed to the NeXT interface, users will not select a time zone, and then pick among a list of names of daylight zones: they will be able to select the daylight zones directly, either by picking a location on a map or selecting a nearby city from a list.[15]

Note that local times are not continuous, and may overlap. E.g. October 31, 1973 might have two local times for times in between 1:00 am and 2:00 am, while April 21 had no times corresponding to between 1:00 am and 2:00 am. Zones are not specific to language: language-specific formatting is handled by the DateTimeFormat.

Daylight rules can also be set to be relative to solar time instead of mean solar time. (With the former, noon is always when the sun crosses the meridian: mean solar time can vary from this by some minutes.) When this is the case, leap seconds are added or removed almost every day at the starting transition time.

Calendar

Given a location and a daylight zone, a Calendar maps from a GMT datetime to a range of date fields and back. Examples of calendars are Gregorian, Japanese, Arabic, Julian, etc. Date fields include: era, year, month, day, hour, minute, second, zone, dayOfWeek, weekOfYear, dayOfYear,...

Calendars also return the range (maximum and minimum values) for any given field in all dates, and the range for any given field with a specific date. They also provide for field toggling, where fields can be incremented or decremented, but wrap around: e.g. toggling the month upward in the date "Dec 29, 1989" results in "Jan 29, 1989". When toggling, other fields remain unaffected, where possible—toggling the month upward in the date "Jan 29, 1989" from January to February cannot be done without changing the day of the month.

---

14 Greenwich Mean Time. Actually, we will be using UTC2 (Coordinated Universal Time) in our standard calendar calculations. UTC2 adds leap seconds during some years to account for variations between Universal Time and International Atomic Time (TA1). UTC2 includes corrections for Chandler wobble and for seasonal changes in the Earth's rotation rate.

15 Some systems have time servers to make sure that computers share the same time (hopefully our OS will have this service). One problem with these systems is that users can't easily set their own clock to something different than the net time. By using a different standard time offset, this can be done.

The field ranges and toggling can be used in the human interface for setting dates and times (e.g. toggling as in the Alarm Clock). A view will be provided for editing time (there are two important variants: absolute and elapsed time).

Calendars are not table-driven, but they may require some persistant data storage for efficiency (e.g. the Arabic Lunar calendar on the Mac has to compute sunset and phases of the moon, so it uses a persistant caching mechanism). Calendars are not specific to language: language-specific formatting is handled by the DateTimeFormat.

Calendars must be set with the current location. Different fields of the calendar can then be gotten and set. The calendar keeps information as to the sequence that fields were set in. Whenever a field is gotten from the calendar, then the information in the calendar is validated. Any of the fields may change in value: the most recently set fields have priority.

For example, suppose that we set the dayOfWeek field, then the year, month and day, then get the dayOfWeek. The dayOfWeek field will be altered to be consistent with the year, month and day, since they were set later. If we then set the dayOfWeek and weekOfYear fields, and get the day field, then it will be generated. The same holds true for converting to internal time (seconds since Jan 1, 1904): if the internal time field is set, and the day field is gotten, then the day corresponding to that time will be generated; if that day field is reset and the internal time is gotten, then it will be regenerated.

A service is available for the use of calendars which given a location, takes a datetime and divides it into three fields: the integral number of days (since the datetime origin), the integral number of seconds in that day, and the remaining fractional seconds. Three options are available: solar time, mean time, and coordinated mean time (using leap seconds). These three fields can also be combined back into a datetime.

We also provide service routines for use by calendars. These include calculating leap-second conversions, sunset times and new moon time & dates.

DateTimeFormat

A datetimeformat contains mappings from date field values to textual representations. The mappings can have variations, such as abbreviated month names, or 2/4 digit years. For example, text field values may include text such as "Sept." "September", "Oct.", "October"; "Friday",.... Any field can have text representations: for example, on some calendars there are names for the days of the month. A datetime format is specific to a given language and calendar. When a date format is used to form a complete date, then the format for each field can be specified:

1. numeric (with various number formats: plain, zero padded, ordinals, outline [e.g. Roman Numerals], <length>; see NumberFormats)

2. text (plain, abbreviated).

The formatted fields are substituted into a text date pattern using pattern substitution. If no date field text is available, then numbers will be used. (The following example illustrates the substitution, but is not necessarily the method we will use. The substitution pattern will contain some data indicating options on the field format):

Pattern:	**"hour:minute ampm zone on dayOfWeek, month day, year zone era"**
hour:	number-only, mod 12, one-based
minute:	number-only, zero padded
ampm:	
zone:	abbreviated
dayOfWeek:	
month:	abbreviated
day:	ordinal

year:
era: abbreviated
Date: 1,493,527,842.000 seconds
Result: "10:08 am PST on Sunday, Oct. 13th, 1958 A.D."

A newly-created format will have simple numerical mappings and a default pattern based on the system language. Additional text names and abbreviated text names can be added for any field values, and the pattern can be changed. The text is of type TStyledText for generality. (Note: the field text might be common to many different patterns, but this model forces them to be stored separately. An alternative architecture breaks patterns and DateTimeFields into separate entities, with the latter shared).

DateTimeFormats will also scan text for date and time, matching against a specific pattern. The scan will stop at the end of the text, end of the pattern or first inappropriate element. The length used by the scan will be returned, as well as any fields found. The scanning will match exactly if possible. Otherwise heuristics will be used, including:

1. skipping over non-letter/digit elements
2. using calendar field ranges to determine suitable matches: 1958/12/23 => y,m,d
3. using the order of elements in the pattern to guess at the scanning order.

A number of standard date formats will be defined for each system. These formats will have absolute ids (names), but will be localized to different languages. That way, if you use a date format by id, then it will be represented in a similar fashion on other localized systems. (E.g. shortDateFormat will appear as "10/21/89" in America, and "21.10.89" in Germany.) Non-standard formats have an associated standard format for display on a foreign system. (E.g. The user creates a new format with the associated longDateFormat, and uses it for an element in a spreadsheet: "Tuesday, March 4th". When he sends it to Germany, it can be left as in English or displayed in a readable German format: 4. März 1989. Notice that the order and selection of fields is not the same in these two languages.

Numbers

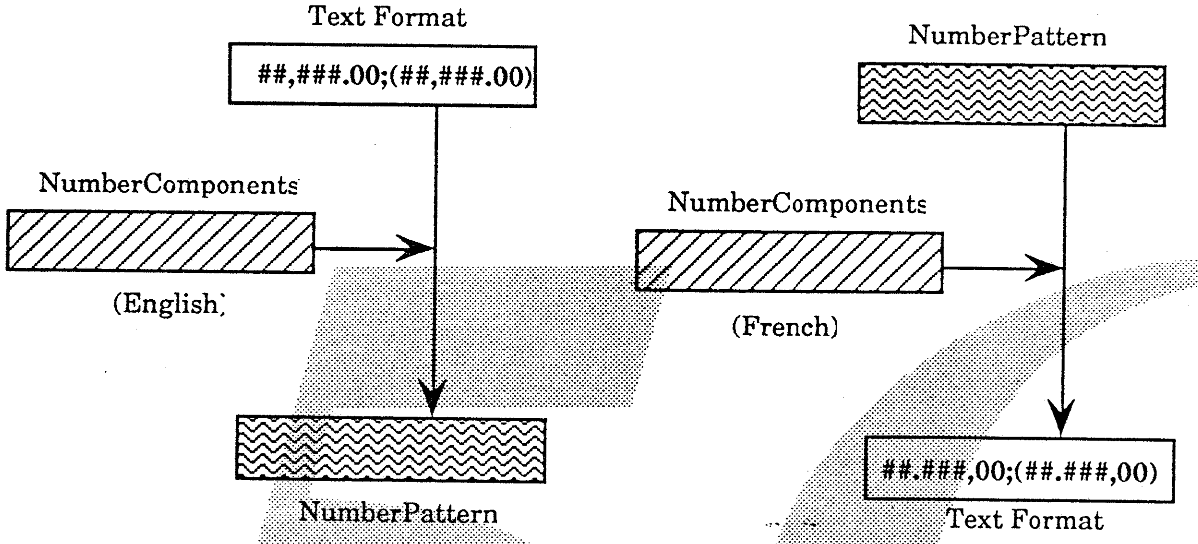
Number formatting involves converting a binary number to a textual representation under programmatic control. Number scanning involves taking a localized number format record and a piece of text, and returning a number and the corresponding text range that was used. Examples of this are provided by the corresponding C routines, although these routines are not international and do not meet the need of more sophisticated clients such as spreadsheets or outliners. These programs need explicit formatting of numbers with thousands-separaters and special formatting, such as $10,000.00 for 100000, ($10,000.00) for -10000, etc.

Number formats include integers, decimals, outline numbers (e.g. Roman numerals, letter numerals), and ordinals (1st, 2nd,...). We will provide both formatting and scanning of numbers based on local conventions (e.g. 5,280.0 vs. 5.280,0). A superset of the functionality on the Mac is provided, whereby pattern matching allows users to get Excel-style functionality while being compatible internationally. Number formatting will also include outline numbers, such as Roman numerals, lettering, etc., and their equivalents in different languages. In general, these will be code based, although tables of localized number components (decimal separator, etc.) will be available.

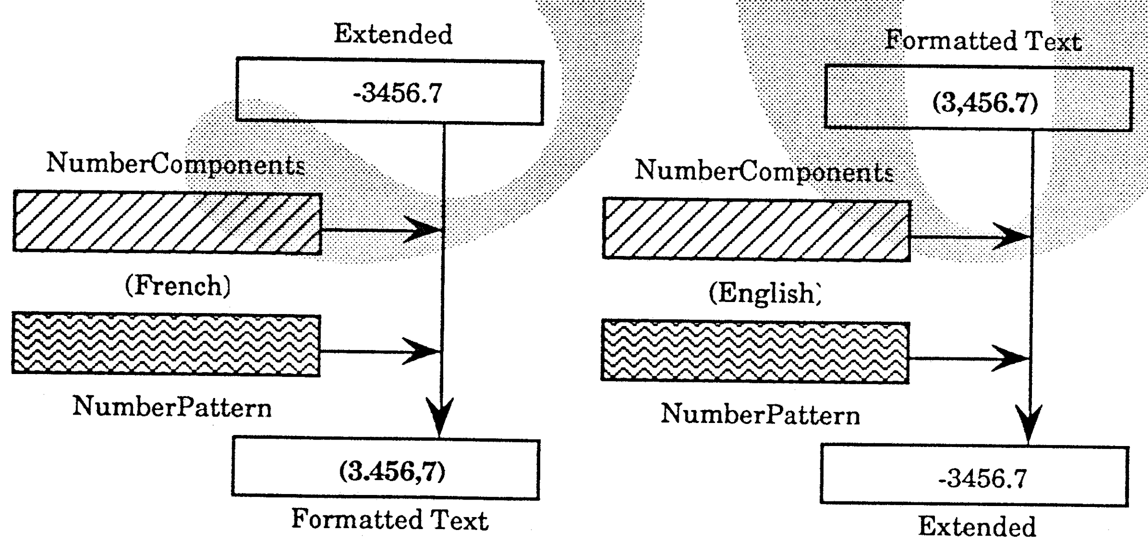
Number scanning will provide not only for the standard Western numerals (decimal 0..9), but also for more unusual systems such as Chinese numbers and outline numbers. Not that in Unicode that the numbers that have significantly different shapes in different scripts have different codes: this is accounted for in both scanning and formatting.

The formatting process first involves producing a number format pattern. This is done by using a localized NumberComponents object to convert a text string as might be supplied by the user, such as "###,###.00" to a NumberPattern. This NumberPattern can also be converted back to a

text string using a NumberComponents, which may be different. For example, converting the above NumberPattern using a French NumberComponents will result in "###.###,00". NumberPatterns may have multiple parts, allowing different formats for positive numbers, negative numbers, zero, positive infinity, negative infinity and NANs (as on the Mac, we will be depending on SANE conversion routines internally).



By using a NumberComponents and a NumberPattern, a client can format extendeds as text, as on the left. Clients can also scan text to extract numbers, as on the right.The scanning process is more forgiving than the format, so that the number 123456.3 can be scanned by the above NumberPattern using an English NumberComponents to produce the correct answer.



Unicode

This document discusses issues and design considerations involved in the allocation of character codes for the standard Apple sixteen-bit character set (Unicode). This character set is designed to be an efficient, internal process code. It is also a foundation for conversions to and from different interchange formats (MS DOS, current Macintosh, ISO 8859, etc.).

Background

The Unicode project at Apple began as a quest to remedy the most serious flaw in the architecture of multilingual text handling on the Macintosh: the overloading of the font mechanism to encode character semantics and properties and the use of multiple, inconsistent character encodings based on conflicting national standards. Unicode envisioned a uniform method of character identification that would be more efficient and general than the Macintosh script systems and reduce dependence on script (i.e. font)-specific software for the display and editing of all text. This uniform model of text will meet two main goals:

- Elimination of special-case system and application code for dealing with multiple character encodings, thereby speeding up the process of localization and reducing testing for applications and system software.
- Making a larger range of characters available to meet the requirements of professional quality typesetting and desktop publishing.

As a result of our research, we concluded that the new Apple character code standard should meet the requirements of:

- **Completeness.** There should be enough bits to easily encompass all characters that will ever be used in general text interchange. A census of the world's scripts and character code standards showed that all scripts in common use can easily fit within a full sixteen bits.[16] Proposals for 24 and 32 bits are only necessary if the code space is wasted by duplicate codes and padding for control codes.
- **Efficiency.** Plain text, composed of a sequence of fixed-width characters, provides an extremely useful model. It's simple to parse: to identify characters, software does not have to maintain state, look for special escape sequences, or search forward or backward through the text. Unicode maintains the simplicity and efficiency of this model and extends it to encompass the writing systems of the world.

  The character encoding method should be independent of methods of compression. For efficient sorting, searching, display, and editing of text, a fixed character code size is preferable to the more complex run-length encodings or mixed 8/16 bit codes that are in current use on many machines. Text compression is important, but need not be defined by the character code standard. There are many different ways to compress text depending on the particular application.

---

16 The initial release of Unicode will contain approximately 25,000 characters of all the world's major scripts, including some 18,000 unique Han characters defined by industry standards in China, Japan, Korea, and Taiwan. This is more than sufficient for modern communication, including such classical languages as Greek, Hebrew, Latin, Pali, Sanskrit, and literary Chinese that may be required by literate non-specialists. Obsolete scripts such as cuneiform, runes, hieroglyphs and additional Han characters used in more specialized research will be added as required.

Alternative Standards

Not wanting to duplicate effort, we also considered the efforts of other companies and international standards bodies in the development of a universal character code set. There is currently one other general multibyte character encoding under development, ISO DP 10646.

The main difference in goals between the Unicode and ISO DP 10646 is that the latter is designed to maximize transmissibility by current products, while Unicode is designed to serve as an efficient 16-bit internal process code for current and future products.

The heart of DP 10646 is the two-byte "Basic Multilingual Plane" (BMP). To maximize transmissibility, DP 10646 reduces the space for graphic characters in the BMP by 29,055 codes to avoid conflict with 8-bit control codes. The remaining graphic character space is further reduced by separately encoding the Han characters (Chinese, Japanese, and Korean logographs) by language, so that 22,594 cells are allotted for the approximately 10,050 unique characters in GB2312-80, JIS X 0208, and KSC 5601. This leaves no room in the BMP for the traditional Han characters used in Hong Kong, Taiwan, and China, or for existing and proposed extensions to the GB and JIS standards. DP 10646 further reduces the BMP by reserving 6,336 codes for glyphs.

A 16-bit code space potentially supports 65,536 characters. But as a result of its design, *DP 10646 provides only about 17,600 unique graphic character codes in the Basic Multilingual Plane.* Since this is insufficient for general multilingual processing, DP 10646 requires a 32-bit encoding, with various methods of selecting subsets of 8, 16, and 24 bits, and shift sequences for addressing characters outside of these subsets.

Methods & Status

Apple and Xerox[17] joined efforts several years ago to come up with the original design of Unicode. Xerox and Apple were soon joined in an informal consortium by representatives of other companies, including Claris, Metaphor, Microsoft, NeXT, Sun, the Research Libraries Group, and others. This cooperation with other companies allowed us to gain the benefit of their contributions in different areas, and increased the opportunities for direct interchange in the future. [18]

Unicode Draft 1 was issued in September 1989, and contained a preliminary repertoire of characters. Unicode Draft 2 is due on March 23, and finalizes the repertoire and codes for all but the Han characters. It includes alphabetic scripts, 1,000 symbols, 18,000 Han characters used in Chinese, Japanese, and Korean, and room for up to 4,096 user characters. Unicode Version 1.0 is due on June 1, and finalizes the Han character repertoire and codes. Additional characters will be added in subsequent versions of Unicode.

We are also active in the ANSI X3L2 multi-byte character encoding committee which is the US national body dealing with ISO 10646. Our goals in that effort are to:

- ensure that Unicode will meet the conventions of an international standard,
- influence the 10646 design in the direction of Unicode (failing that, ensure that the 10646 design does not preclude transcoding with Unicode),

---

17 Xerox is also very active in the area of multilingual operating systems (though not in terms of units), and had realized that their character encoding methods could also be significantly improved.

18 Xerox has been particularly helpful in the design of Unicode and in printing the draft Unicode charts.

- encourage other countries and companies to consider the advantages of the Unicode approach over 10646.

We were able to get approval from the ANSI X3L2 for including the major Unicode principles within 10646 (Document X3L2/89-195: "Proposed Modifications to ISO DP 10646", a *U.S. National Body Position*). The principal motivation for the ANSI support of this proposal was to avoid two, separate multi-byte standards. The initial international reception for this change, however, has not been promising.

Design

Developing a character set consists of two steps: selecting the repertoire of characters to go into the set and assigning codes to the repertoire. Many coding standards have run into problems because they attempt to mix formatting codes, compression methods, or glyph encodings into their standard.

Unicode *Characters* are fixed sixteen-bit identifiers, representing primarily, but not exclusively, the letters, punctuation, and other signs that compose natural language text, including foreign language, math, science, and other technical documentation. Characters reside only in the machine, as strings in memory or on disk, in what we call the *backing store*. In contrast to characters, *Glyphs* exist in a font and appear on the screen and paper as particular instances of one or more backing store characters. End users see glyphs, not characters.

The process of mapping from characters in the backing store to glyphs is one aspect of text *rendering*. The final appearance of rendered text is dependent on context (neighboring characters in the backing store), variations in typographic design of the fonts used, and formatting information (point size, superscript, subscript, etc.). The result on screen or paper can differ considerably from the archetypical "shape" of a character.

In allocating character codes, the *glyph A* that we see on the screen must not be confused with the *character A* in the backing store. However, for convenience Unicode generally chooses a single glyph shape to represent characters in the code tables.

Unicode deals only with character codes. Glyph shape and glyph identifier assignments are up to individual font vendors or the glyph identifier standards such as that being developed by the Association for Font Information Interchange.

The Unicode Repertoire

Selecting the repertoire is not a science; there are too many conflicting usages to derive hard and fast rules. However, Unicode uses the following general principles.

- **Constant-Width, 16-bit Encoding.** Just as with simple ASCII text, developers need not parse text for shift sequences to determine character boundaries or encoding. The elimination of the requirement for shift sequences also permits efficient random access of characters.

  Unicode is no different from any other binary data and can be compressed to reduce storage, transformed for transmission over 7 bit or 8 bit lines, or transcoded to and from other character encodings. There are many algorithms available for performing compression and transformation (e.g. Unix btoa which converts 32 bits to five 7-bit ASCII characters): Unicode itself does not currently specify any particular methods.

- **Full Encoding.** Unicode reserves room for a limited number of control codes; aside from these it uses the full 16-bit range to represent over 65,000 graphic characters.

- **Complete Encoding.** Unicode provides character codes for characters used in the computing and the publishing industry around the world. It draws from a base of national and international character encodings, including: ANSI Z39.47-1985 (bibliographic Roman), ISO 5426-1983 (bibliographic Roman), ISO 5427 (bibliographic Cyrillic), ISO 5428-1964 (bibliographic Greek), ISO 6438 (extended Roman for African languages), ISO 6861

(Glagolitic, Old Cyrillic, and Romanian Cyrillic), ISO 6862 (mathematical symbols), ISO 6937 (Western European Roman script), ISO 8859/1-8 (8-bit sets for all European Roman, Greek, Cyrillic, Arabic, and Hebrew), ISCII (India), GB 2312-80 (China), JIS X 0208 (Japan), KS C 5601-87 (Korea), and CNS 11643 (Taiwan). It also includes de facto company standards, such as the Fujutsu extensions to JIS.

- **Preservation of Base Distinctions.** For ease of transcoding, Unicode preserves differences found in the base standard encodings mentioned above. For example, Greek "Omicron" (O) is distinguished from English "Oh" (O) because they are distinguished in ISO 8859/7.

- **Pure Character Encoding.** Other than what is required to preserve base distinctions, Unicode does not attempt to encode features such as language, font, size, positioning, glyphs, et cetera. For example, it does not preserve language as a part of character encoding: Chinese "zi" (字), Japanese "ji" (字) and Korean "ja" (字) are all represented as the same character code, as are French "i grecque" (Y), German "ypsilon" (Y), and English "wye" (Y).

  The distinction between Unicode and other forms of data (ASCII, pictures, etc.) is also the function of a higher-level protocol (e.g. text classes & layout) and not specified by Unicode itself.

- **Dynamic composition.** Unicode allows dynamic composition of accented forms or static composed forms. Common static-form single character codes such as *diaeresisUpperU* "Ü" are allowed for compatibility with current Macintosh and international standards. However, because the process of character composition is open-ended, additional letters with modifying diacritics can be dynamically created from a combination of base letters and diacritic characters.

  Unicode accents and other diacritics used to create composite forms are *productive*. This means that they are *in practice* combined with a large number of base form characters. For example, the *diaeresis* ¨ can be combined with all vowels and a number of consonants in languages using the Roman script. The *slash* used in the Polish *slashL*, however, is of limited use, hence, *slashL* is treated as a single character and not dynamically composited from two characters.

  This extends to non-Roman languages as well. In particular, the infrequent Hangul letters that are not included in the set of precomposed forms are available by using composition.

- **Natural Language Letters.** Unicode limits character code assignments for natural language letters to what native speakers think of as the components (e.g., the basic letters of the alphabet) of their script. For compatibility with existing standards, however, Unicode does separate case variants ("a" and "A") of the same letters.

- **Common Characters.** Unicode encourages the borrowing of characters from other scripts and avoids duplication except in the following cases:

  a. Operators and well-known constants borrowed from Greek (e.g. π or "pi", and Σ or "summation") and other scripts are allocated separate character codes. However, this is not true for Greek letters used as variables.

  b. Apparent style variants[19] with specific usages (e.g. the script form "ɑ" in the International Phonetic Alphabet, which represents a low, back unrounded vowel) are assigned separate character codes.

- **Han Unification.** From the above, it follows that Unicode does not define separate codes for the Han characters (*hanzi, kanji, hanja*) used in Chinese, Japanese, and Korean. In unifying Han characters, Unicode follows these rules:

---

19. As noted above, for each script we assume a single font family and style to represent the archetypical shape of a character. A style variant is any departure from this base shape.

a. The initial repertoire is based on the four major standards for Han characters: GB 2312-80 (China: 6,763 characters), JIS X 0208 (Japan: 6,349 characters), KS C 5601-87 (Korea: 4,888 characters), and CNS 11643 (Taiwan: 13,051 characters), and the approximately 5,000 characters of the proposed extensions to the JIS standard.

b. Character identity is preserved over the combined standards. Where variant forms are given separate codes within one standard, they are also kept separate within Unicode.[20] This guarantees that there will always be a mapping between Unicode and target national standards.

c. In determining whether or not to unify variant forms across standards, Unicode follows the guidelines published by JIS.[21] Where these guidelines suggest that two forms constitute a trivial (*wazukana*) difference, Unicode assigns a single code. Otherwise, separate codes are assigned.

- **Digraphs.** Unicode provides digraphs such as Dutch *IJ* for compatibility with existing standards. Otherwise, the software for text processing is assumed to be sufficiently powerful to guarantee correct behavior of simple digraphs (e.g. "ch" in Spanish) in collating, comparison, layout, etc., with the use of zero-width separators as necessary (see below).

Future Expansion and Character Registration

In the first implementation of Unicode we are as thorough as possible in assigning codes for all characters required for the scripts selected. Some scripts have not yet been added and others (the Han characters in particular) will certainly require new characters as they are added to national standards in the future. The symbols selected represent mainly those currently in common use on computer systems or which are found in standards.[22] We do not consider the symbol set to be complete and expect that codes for most application-specific symbols will be assigned in the private user space. However, a registration process must be established to add symbols and other characters for common use to the public symbol space. We would like to leverage the efforts of other companies to establish a general Unicode consortium character registry.

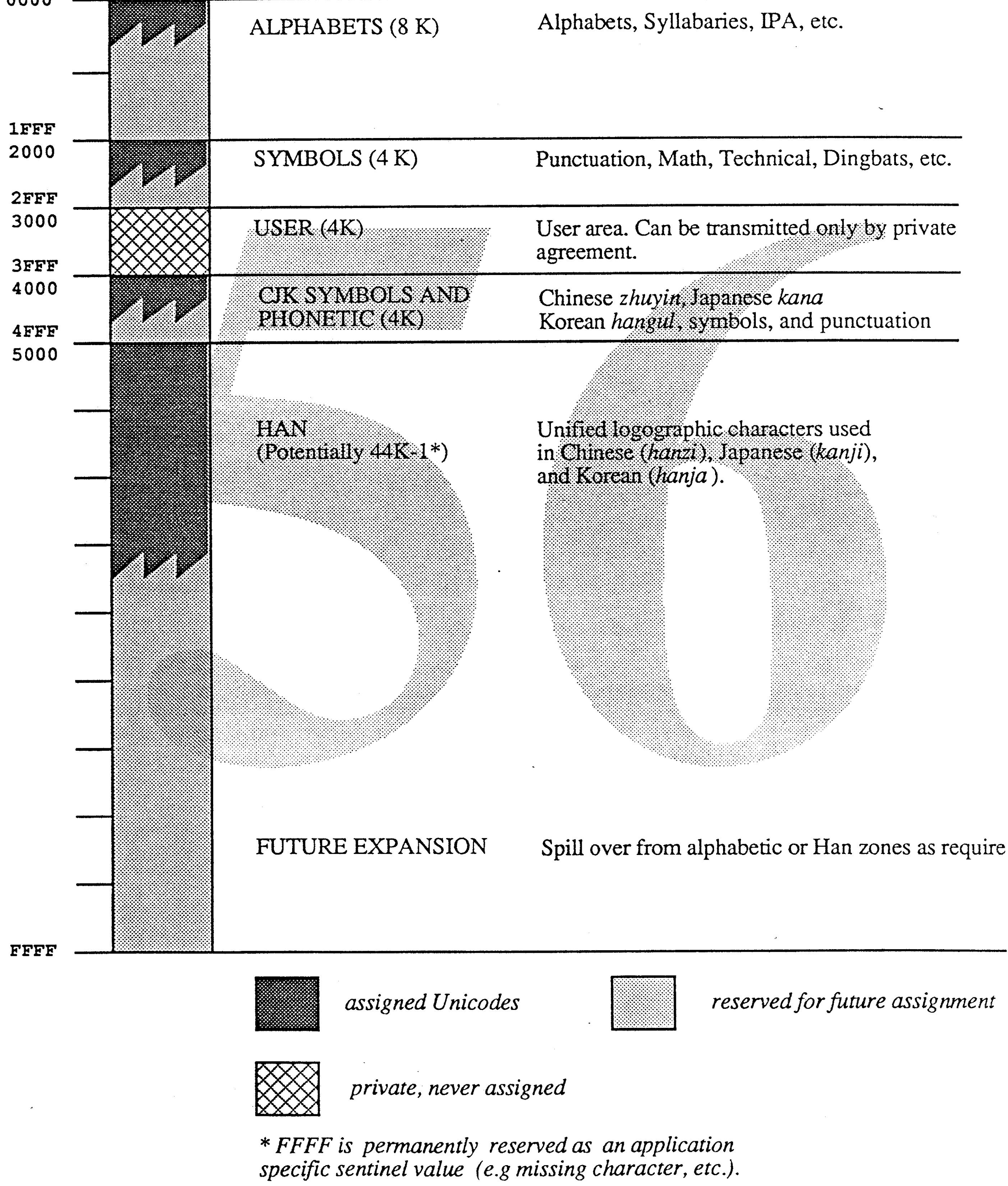
---

20 For example, JIS assigns six characters (23-85, 49-88, 49-89, 49-90, 49-91, 78-63) for variants of the character ken "sword." Unicode preserves all of these.

21 These guidelines are found in the Japan Industrial Standard *Jouhou koukan you kanji fugoukei* (Code of the Japanese (sic) Graphic Character Set for Information Interchange) C 6226-1983. §3.4 *Kanji no itaiji no toriatsukai* (The handling of variant Han characters). Though written with Japanese usage in mind, they are general enough to be applied across all three languages.

22 Many of the symbol characters, such as the Roman numerals, have been added only because they exist in some standard.

Unicode Codespace Allocation



** FFFF is permanently reserved as an application specific sentinel value (e.g missing character, etc.).*

Code Assignment

Unicode is divided into five zones: Alphabetic and other scripts having relatively small character sets, Symbols, User characters, Auxiliary characters for Chinese, Japanese, and Korean, and Han characters. The alphabetic zone covers alphabetic or syllabic scripts such as Roman, Cyrillic, Greek, Arabic, Devanagari, Thai, etc. The symbol zone includes a large variety of characters for punctuation, mathematics, chemistry, dingbats, etc. The user zone (about 4,000 code points) is used for defining user- or vendor-specific graphic characters. The Chinese, Japanese, and Korean auxiliary characters include punctuation, symbols, *kana*, *zhuyinfuhao*, and single and composite *hangul*. The Han characters subset provides for over 44,000 logographic characters common to Chinese, Japanese and Korean. Unicode Draft 1 currently covers the complete repertoire of 16,400 unique Han characters defined in the GB, CNS, JIS and KSC standards. Unicode version 1.0 will extend coverage to include approximately 18,000 characters.

- Unicode is "fully coded." Only the first thirty-two codes ($0000-$0020) are reserved specifically as "control codes". $FFFF is also reserved for internal use (e.g. as a sentinel) and should not be transmitted or stored. All other code points can be freely used for graphic characters. Null ($0000) can be used as a string terminator as in the C language. 4K of user space has been allocated in the range $3000-$3FFF. There is no mechanism for extension into a larger code space, so programmers are not subject to the need to test every character for escape sequences.
- One change since draft 1.0 is that we have decided to follow currently existing standards for the relative order of characters within a script, where there is a single accepted standard for that script. In one notable case, we have used ASCII (characters $0000 through $007F) and ISO 8859-1 (characters $00A0-$00FF) standards for the first 256 characters.
- It is common for programmers to confuse coding order with collation order. However, in no language that we are aware of are raw character codes alone are sufficient for collating. Certainly naïve use of ASCII does not work, where 'zoo' sorts before 'Apple'! The same holds true for upper/lower casing, which is also language dependent. Because of these facts, the Unicode character code assignment does not go out of its way to correlate character encoding with collation or case.
- Commonly used sets of characters or characters with common characteristics are located together in a contiguous range. For example, Arabic script characters used in Persian, Urdu, and other languages but which are not included in ASMO 662 are grouped closely following the basic ASMO set.

	Section	Start Code	No. Assigned
Alphabets	Control	0000	32
	Roman	0020	437
	Phonetic	0250	72
	Modifier Letters	02B0	39
	Diacritics	0300	60
	Greek	0370	125
	Cyrillic	0400	188
	Georgian	0500	39
	Armenian	0530	83
	Hebrew	0590	85
	Arabic	0600	167
	Ethiopian	0700	352
	Devanagari	0900	100
	Bengali	0980	84
	Gurmukhi	0A00	67
	Gujarati	0A80	74
	Oriya	0B00	75
	Tamil	0B80	60
	Telegu	0C00	78
	Kannada	0C80	78
	Malayalam	0D00	77
	Sinhalese	0D80	81
	Thai	0E00	87
	Lao	0E80	65
	Burmese	0F00	72
	Khmer	0F80	85
	Tibetan	1000	74
	Mongolian	1080	104
Symbols	Punctuation	2000	46
	Super/Subscripts	2050	33
	Number Forms	2080	51
	Currency	20D0	10
	Letterlike Symbols	2100	45
	Symbol Diacritics	2150	17
	Enclosed Alphanumerics	2180	112
	OCR	2200	13
	Components	2220	28
	Mathematics Operators	2240	174
	Miscellaneous	2300	64
	Control Character Pictures	2360	45
	Arrows	23A0	72
	Geometric Shapes	2400	65
	Basic Dingbats	2460	119
	Mosaics	2500	189
	Borders	25C0	21
	Forms	2600	117
CJK Aux.	CJK Symbols	4000	50
	Hiragana	4040	88
	Katakana	40A0	87
	Zhuyin Fuhao	4100	38
	Hangul Letters	4130	94
	Parenthesized Letters	4300	88

	Encircled Letters	4380	98
	Squared Units	4400	175
	Hangul Syllables	4500	2609
Han	Han Logographs	5000	18,000

Details

The following are particular points about the design of Unicode that specify details that are necessary to determine correct use and interchange.

Paragraph/Line Separators

Unicode does not specify a particular character for a paragraph separator or a line separator (a line separator causes a line break (e.g. '
') but does not start a new paragraph, so the interparagraph spacing and paragraph indent are not applied. Apple will probably continue to use CR for paragraph separator and LF for line separator. (Note that these are separators, not terminators or initiators: that is, a paragraph separator servers only to separate two paragraphs: it need not occur with every paragraph.)

Specific Characters

In many cases, current standards include a generic character instead of a number of specific characters that are used in printing. Examples of this are the generic single & double quotes, period, dash, space, etc. Unicode includes these characters, but also includes the disambiguated characters: right & left quotes; abbreviation period, sentence period, decimal period; em-dash, en-dash, minus, hyphen; em-space, en-space, hair-space, zero-width space, etc.

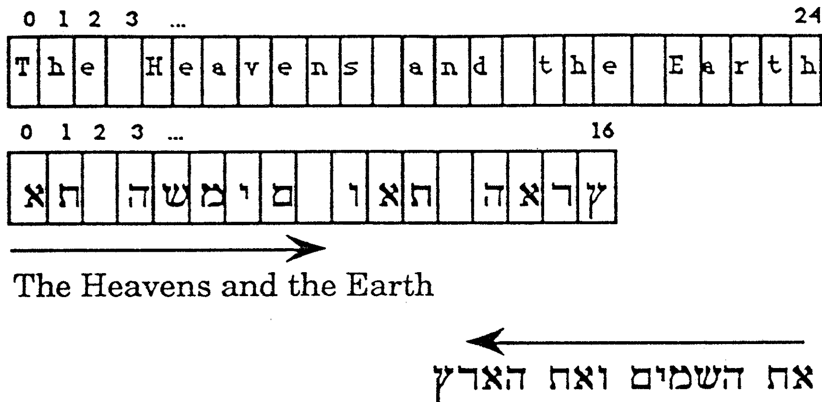
There are several varieties of zero-width spaces: the standard one is a word-break space, used to add soft-word breaks in languages without word-spaces. Additionally, there are two spaces which can be used in controlling cursive forms of characters, the zero-width joiner and zero-width non-joiner. There are also zero-width directional spaces, the right-to-left zero-width space and the left-to-right zero-width space.[23] All of these properties are orthogonal: the word-break space does not affect joining or direction; the joiners do not cause a word-break or have a direction; the directional spaces do not cause word-breaks or affect joining.

Ordering of Character Sequences

Unicode text is stored in a single "logical" order in the backing store. Logical or backing store order is based on the chronological order in which text is spoken. English and Hebrew strings are shown below first as they would be ordered first in logical order, then in display order.

---

23. We will probably not use these in Pink to control direction, although we will interpret them properly if we receive interchanged text that includes them.



In the example, the first character of each backing store string is at position 0, but the displayed text is in the correct writing order. Logical order is independent of display order. In the text ordered for display, the glyph that matches the logical order character zero of the English text is at the left. The logical start character of the Hebrew text, however, now corresponds to the glyph closest to the right margin. The succeeding Hebrew glyphs are laid out to the left. Unicode assumes that higher level software will be responsible for rendering groups of backing store characters in correct display order. Three cases require explicit mention:

a. Non-spacing characters (accents marks in the Greek and Roman scripts, vowel marks in Arabic and Devanagari, etc.) do not appear linearly in the final rendered text. In a Unicode string all such characters follow the base character which they modify, or the character after which they would be articulated in "phonetic" order (e.g. Roman "ã" is stored as "a~" when not a static form).

b. All scripts are stored in logical order. This applies even when mixing scripts whose dominant direction may be left-to-right (Greek, Roman, Thai, etc.), right-to-left (Arabic, Hebrew, etc.), or vertical (Mongolian, etc.). Properties of directionality inherent in each character determine the correct display order of text which contains only a single script. Displaying a mixture of such scripts, however, is a product of higher-level formatting information, as is the choice of font, size, style, and position (superscript, subscript), etc. Higher-level line and page layout software may also wish to override basic character directionality.

c. Characters such as the medial form of the "short i" (ि) in Devanagari are displayed before the characters that they logically follow in the backing store.

Sample Code Pages

The following are sample pages of the Unicode code charts. They are followed by a sample page of the Han character cross-reference. The cross reference is in radical-stroke order, with the characters from each of the four base Han standards listed together. The numbers underneath each character are the code points in the respective base standards.

Unicode 0 0 _ _ Hex

Draft of 2/15

	Control		ASCII						Misc.		Latin1					
	000	001	002	003	004	005	006	007	008	009	00A	00B	00C	00D	00E	00F
0	NUL	DLE	SP	0	@	P	`	p	–	≠	NB SP	°	À	Ð	à	ð
1	SOH	DC1	!	1	A	Q	a	q	—	≤	¡	±	Á	Ñ	á	ñ
2	STX	DC2	"	2	B	R	b	r	‘	≥	¢	²	Â	Ò	â	ò
3	ETX	DC3	#	3	C	S	c	s	’	≈	£	³	Ã	Ó	ã	ó
4	EOT	DC4	$\|$	4	D	T	d	t	“	∂	¤	´	Ä	Ô	ä	ô
5	ENQ	NAK	%	5	E	U	e	u	”	∆	¥|¥	µ	Å	Õ	å	õ
6	ACK	SYN	&	6	F	V	f	v	‚	ƒ	¦	¶	Æ	Ö	æ	ö
7	BEL	ETB	'	7	G	W	g	w	„	∏	§	·	Ç	×	ç	÷
8	BS	CAN	(	8	H	X	h	x	•	π	¨	¸	È	Ø	è	ø
9	HT	EM	)	9	I	Y	i	y	◊	Ω	©	¹	É	Ù	é	ù
A	LF	SUB	*	:	J	Z	j	z	…	Œ	ª	º	Ê	Ú	ê	ú
B	VT	ESC	+	;	K	[	k	{	†	œ	«	»	Ë	Û	ë	û
C	FF	FS	,	<	L	\	l	|	‡	∫	¬	¼|¼	Ì	Ü	ì	ü
D	CR	GS	−	=	M	]	m	}	‰	∑	−	½|½	Í	Ý	í	ý
E	SO	RS	.	>	N	^	n	~	∞	™	®	¾|¾	Î	Þ	î	þ
F	SI	US	/	?	O	_	o	DEL	√	Ÿ	¯	¿	Ï	ß	ï	ÿ

	Generic Diacritical Marks							Greek								
	030	031	032	033	034	035	036	037	038	039	03A	03B	03C	03D	03E	03F
0	◌̀	◌̐	◌̠	◌̰				◌ͅ		ΐ	Π	ΰ	π	ϐ	ϡ	
1	◌́	◌̑	◌̡	◌̱				ʽ		Α	Ρ	α	ρ	ϑ	ϡ	
2	◌̂	◌̒	◌̢	◌̳				ʼ		Β		β	ς	ϒ	Ϣ	
3	◌̃	◌̓	◌̣	◌̴						Γ	Σ	γ	σ	ϓ	ϣ	
4	◌̄	◌̔	◌̤	◌̵					◌̓ͅ	Δ	Τ	δ	τ	ϔ	Ϥ	
5	◌̅	◌̕	◌̥	◌̶					◌̈́	Ε	Υ	ε	υ	φ	ϥ	
6	◌̆	◌́	◌̦	◌̷					Ά	Ζ	Φ	ζ	φ	ϖ	Ϧ	
7	◌̇	◌̀	◌̧	◌̸						Η	Χ	η	χ	;	ϧ	
8	◌̈	◌̑	◌̨	◌͠					Έ	Θ	Ψ	θ	ψ	ʹ	Ϩ	
9	◌̉	◌́	◌̩	◌͞					Ή	Ι	Ω	ι	ω	͵	ϩ	
A	◌̊	◌̄	◌̪	◌͝					Ί	Κ	Ϊ	κ	ϊ	Ϛ	Ϫ	
B	◌̋	◌̚	◌̫	◌͡						Λ	Ϋ	λ	ϋ	ϛ	ϫ	
C	◌̌	◌̜	◌̬	◌͜					Ό	Μ	ά	μ	ό	Ϝ	Ϭ	
D	◌̍	◌̝	◌̭							Ν	έ	ν	ύ	ϝ	ϭ	
E	◌̎	◌̞	◌̮						Ύ	Ξ	ή	ξ	ώ	Ϟ|Ϙ	Ϯ	
F	◌̏	◌̟	◌̯						Ώ	Ο	ί	ο		ϟ|ϙ	ϯ	

Unicode 0 5 _ _ Hex

	Georgian			Armenian						Hebrew						
	050	051	052	053	054	055	056	057	058	059	05A	05B	05C	05D	05E	05F
0		ო	წ		Հ	Ր		հ	ր	א	נ	◌ְ	׀		◌֒	◌֭
1	ა	პ	ჭ	Ա	Ձ	Ց	ա	ձ	ց	ב	ס	◌ֱ	◌ׁ	◌֑	◌֓	
2	ბ	ჟ	ხ	Բ	Ղ	Ւ	բ	ղ	ւ	ג	ע	◌ֲ	◌ׂ	◌֖	◌֔	
3	გ	რ	ჴ	Գ	Ճ	Փ	գ	ճ	փ	ד	ף	◌ֳ	׃	◌֖	◌֕	
4	დ	ს	ჯ	Դ	Մ	Ք	դ	մ	ք	ה	פ	◌ִ	‹	◌֗	◌֙	
5	ე	ტ	ჰ	Ե	Յ	Օ	ե	յ	օ	ו	ץ	◌ֵ	◌֘	◌֗	◌֘	
6	ვ	ჳ	ჵ	Զ	Ն	Ֆ	զ	ն	ֆ	ז	צ	◌ֶ	◌֑	◌֢	◌֡	
7	ზ	უ	ჶ	Է	Շ		է	շ		ח	ק	◌ַ		◌֬	◌֒	
8	ჱ	ფ		Ը	Ո		ը	ո		ט	ר	◌ָ		◌֥	◌֠	
9	თ	ქ		Թ	Չ	ՙ	թ	չ	։	י	ש	◌ֹ	׳	◌֮	◌֩	
A	ი	ღ		Ժ	Պ	՚	ժ	պ		ך	ת	◌ֹ	״	◌֣	◌֓	
B	კ	ყ	፡	Ի	Ջ	՛	ի	ջ		כ		◌ֻ	₪	◌֤	‹	
C	ლ	შ		Լ	Ռ	՜	լ	ռ		ל		◌ּ		◌֞	◌֩	
D	მ	ჩ		Խ	Ս	՝	խ	ս		ם	װ	◌ֽ		◌֜	◌֒	
E	ნ	ც		Ծ	Վ	՞	ծ	վ		מ	ױ	־		◌֗	◌֒	
F	ჲ	ძ		Կ	Տ	՟	կ	տ		ן	״	◌ֿ		◌֯		

Unicode 04__ Hex

	Cyrillic									Extended Cyrillic						
	040	041	042	043	044	045	046	047	048	049	04A	04B	04C	04D	04E	04F
0		А	Р	а	р		Ѡ	Ѱ	Ҁ	Ґ	Ҡ	Ұ	Ӏ			
1	Ё	Б	С	б	с	ё	ѡ	ѱ	ҁ	ґ	ҡ	ұ				
2	Ђ	В	Т	в	т	ђ	Ѣ	Ѳ	҂	Ғ|Ғ	Ң|Ң	Ҳ				
3	Ѓ	Г	У	г	у	ѓ	ѣ	ѳ	◌҃	ғ|ғ	ң|ң	ҳ				
4	Є	Д	Ф	д	ф	є	Ѥ	Ѵ	◌҄	Ҕ	Ҥ	Ҵ				
5	Ѕ	Е	Х	е	х	ѕ	ѥ	ѵ	◌҅	ҕ	ҥ	ҵ				
6	І	Ж	Ц	ж	ц	і	Ѧ	Ѷ	◌҆	Җ	Ҧ	Ҷ|Ҷ				
7	Ї	З	Ч	з	ч	ї	ѧ	ѷ	◌҇	җ	ҧ	ҷ|ҷ				
8	Ј	И	Ш	и	ш	ј	Ѩ	Ѹ	◌҈	Ҙ|Ҙ	Ҩ	Ҹ				
9	Љ	Й	Щ	й	щ	љ	ѩ	ѹ		ҙ|ҙ	ҩ	ҹ				
A	Њ	К	Ъ	к	ъ	њ	Ѫ	Ѻ		Қ|Қ	Ҫ|Ҫ	Һ				
B	Ћ	Л	Ы	л	ы	ћ	ѫ	ѻ		қ|қ	ҫ|ҫ	һ				
C	Ќ	М	Ь	м	ь	ќ	Ѭ	Ѽ		Ҝ	Ҭ	Ҽ				
D		Н	Э	н	э		ѭ	ѽ		ҝ	ҭ	ҽ				
E	Ў	О	Ю	о	ю	ў	Ѯ	Ѿ		Ҟ	Ү	Ҿ				
F	Џ	П	Я	п	я	џ	ѯ	ѿ		ҟ	ү	ҿ				

	General Punctuation					Supers. & Subs			Number Forms					Currency		
	200	201	202	203	204	205	206	207	208	209	20A	20B	20C	20D	20E	20F
0	En Quad	--	′			⁰	₀	△	ø	I	i	ↀ		₠		
1	Em Quad	—	″				₁	→	θ	II	ii	ↁ		₡		
2	En SP	. Fixed Pnt	‴				₂	←	7	III	iii	ↂ		₢		
3	Em SP	. Anchor Pnt	‵				₃	↔	⅓	IV	iv			₣		
4	3 Em SP	. Hyphen Pt	‶			⁴	₄		⅔	V	v			Kr		
5	4 Em SP	. Decimal	^			⁵	₅		⅕	VI	vi			₤		
6	6 Em SP	. 1000's	‹			⁶	₆		⅖	VII	vii			₥		
7	Fig SP	, Decimal	›			⁷	₇		⅗	VIII	viii			₦		
8	Punct SP	, 1000's	▶			⁸	₈		⅘	IX	ix			₧		
9	Thin SP	·|, Decimal	‰			⁹	₉		⅙	X	x			₨		
A	Hair SP	·|, 1000's	※			⁺	₊		⅚	XI	xi			₩		
B	Zero Width SP	.	#			⁻	₋		⅛	XII	xii					
C	Zero Width Letter	..	♭			⁼	₌		⅜	L	l					
D	Hyphen	‼	♮			⁽	₍		⅝	C	c					
E	NB	‘				⁾	₎		⅞	D	d					
F	Figure	“				ⁿ	ᵢ		1/	M	m					

OCR		Parts		Mathematics Operators											
220	221	222	223	224	225	226	227	228	229	22A	22B	22C	22D	22E	22F
⑀		⎛	⎜	∀	∔	∥	∶	≅	≗	≪	≺	⊌	⊜	⊩	
⑁		⎜	⟍	∁	∕	∦	∷	≆	≘	≫	≻	⊍	⦶	⊪	
⑂		⎝	∠	∃	⋆	∧	∸	≇	≙	≪	≼	⊎	⊞	⊬	
⑃		⎞	╲	∄	∗	∨	∹	≈	≚	≫	≽	⊏	⊟	⊭	
⑄		⎠	╱	∅	∘	∩	∺	≋	≛	≮	≾	⊐	⊠	⊮	
⑅		⎤	─	∇	∙	∪	∻	≌	≜	≯	≿	⊑	⊡	⊯	
⑆		⎬	▁	∊	∛	∬	∼	≍	≝	≰	⊂	⊒	⊢	⊲	
⑇		⎦	〉	⋶	∜	∭	∽	≎	≞	≱	⊃	⊓	⊣	⊳	
⑈		⎡	⎡	∍	∝	∮	∾	≏	≟	≲	⊄	⊔	⊤	⊴	
⑉		⎣	⎣	∈	⊥	∯	∿	≐	≡	≳	⊅	⊕	⊥	⊵	
⌶		⎨	⎤	∉	∟	∰	≀	≑	≢	≴	⊆	⊖	⊦	⊶	
⫲		⌠	⎦	∋	∠	⨍	≁	≒	≣	≵	⊇	⊗	⫞	⊷	
⋈		⎩		∌	∡	∲	∔	≓	≦	≶	⊈	⊘	⊧	⊸	
		⎧		∐	∢	∳	≂	≔	≧	≷	⊉	⊙	⫣	⌖	
		⎭		−	∣	∴	≃	≕	≨	≸	∈	⊚	⊨		
		√		∓	∤	∵	≄	≖	≩	≹	∋	⊛	⫤		

Unicode 40__ Hex

Draft of 2/12

	CJK Symbols				Hiragana						Katakana					
	400	401	402	403	404	405	406	407	408	409	40A	40B	40C	40D	40E	40F
0	〜	【	〳	〠		ぐ	だ	ば	む	ゐ		グ	ダ	バ	ム	ヰ
1	、	】	〴	[illegible]	ぁ	け	ち	ぱ	め	ゑ	ァ	ケ	チ	パ	メ	ヱ
2	。	〒	〵		あ	げ	ぢ	ひ	も	を	ア	ゲ	ヂ	ヒ	モ	ヲ
3	〃	〓	〡		ぃ	こ	っ	び	ゃ	ん	ィ	コ	ッ	ビ	ャ	ン
4	仝	〔	〢		い	ご	つ	ぴ	や	ゔ	イ	ゴ	ツ	ピ	ヤ	ヴ
5	々	〕	〣		ぅ	さ	づ	ふ	ゅ		ゥ	サ	ヅ	フ	ュ	ヵ
6	〆	〖	〤		う	ざ	て	ぶ	ゆ		ウ	ザ	テ	ブ	ユ	ヶ
7	○	〗	〥		ぇ	し	で	ぷ	ょ		ェ	シ	デ	プ	ョ	
8	〈	〘	〦		え	じ	と	へ	よ		エ	ジ	ト	ヘ	ヨ	
9	〉	〙	〧		ぉ	す	ど	べ	ら		ォ	ス	ド	ベ	ラ	
A	《	〚	〨		お	ず	な	ぺ	り		オ	ズ	ナ	ペ	リ	
B	》	〛	〩		か	せ	に	ほ	る	゛	カ	セ	ニ	ホ	ル	
C	「	〝	〪		が	ぜ	ぬ	ぼ	れ	゜	ガ	ゼ	ヌ	ボ	レ	ー
D	」	〞	〫		き	そ	ね	ぽ	ろ	ゝ	キ	ソ	ネ	ポ	ロ	ヽ
E	『	─	〬		ぎ	ぞ	の	ま	ゎ	ゞ	ギ	ゾ	ノ	マ	ヮ	ヾ
F	』	〰	〭		く	た	は	み	わ		ク	タ	ハ	ミ	ワ	

	Bopomofo			Hangul Elements						CJK Mini User Space						
	410	411	412	413	414	415	416	417	418	419	41A	41B	41C	41D	41E	41F
0		ㄐ	ㄠ		ㅀ	ㅐ	ㅠ	ㅰ	ㆀ							
1		ㄑ	ㄡ	ㄱ	ㅁ	ㅑ	ㅡ	ㅱ	ㆁ							
2		ㄒ	ㄢ	ㄲ	ㅂ	ㅒ	ㅢ	ㅲ	ㆂ							
3		ㄓ	ㄣ	ㄳ	ㅃ	ㅓ	ㅣ	ㅳ	ㆃ							
4		ㄔ	ㄤ	ㄴ	ㅄ	ㅔ	(채움)	ㅴ	ㆄ							
5	ㄅ	ㄕ	ㄥ	ㄵ	ㅅ	ㅕ	ㅥ	ㅵ	ㆅ							
6	ㄆ	ㄖ	ㄦ	ㄶ	ㅆ	ㅖ	ㅦ	ㅶ	ㆆ							
7	ㄇ	ㄗ	ㄧ	ㄷ	ㅇ	ㅗ	ㅧ	ㅷ	ㆇ							
8	ㄈ	ㄘ	ㄨ	ㄸ	ㅈ	ㅘ	ㅨ	ㅸ	ㆈ							
9	ㄉ	ㄙ	ㄩ	ㄹ	ㅉ	ㅙ	ㅩ	ㅹ	ㆉ							
A	ㄊ	ㄚ	ㄫ	ㄺ	ㅊ	ㅚ	ㅪ	ㅺ	ㆊ							
B	ㄋ	ㄛ		ㄻ	ㅋ	ㅛ	ㅫ	ㅻ	ㆋ							
C	ㄌ	ㄜ		ㄼ	ㅌ	ㅜ	ㅬ	ㅼ	ㆌ							
D	ㄍ	ㄝ		ㄽ	ㅍ	ㅝ	ㅭ	ㅽ	ㆍ							
E	ㄎ	ㄞ		ㄾ	ㅎ	ㅞ	ㅮ	ㅾ	ㆎ							
F	ㄏ	ㄟ		ㄿ	ㅏ	ㅟ	ㅯ	ㅿ								

	CJK Parenthesized								CJK Encircled							
	430	431	432	433	434	435	436	437	438	439	43A	43B	43C	43D	43E	43F
0	(ア)	(チ)	(ㄱ)	(다)	(一)	(月)	(自)		ア	チ	ㄱ	다	正	左	夜	
1	(イ)	(リ)	(ㄴ)	(라)	(二)	(火)	(資)		イ	リ	ㄴ	라	適	右	企	
2	(ウ)	(ヌ)	(ㄷ)	(마)	(三)	(水)	(特)		ウ	ヌ	ㄷ	마	標	注	医	
3	(エ)	(ル)	(ㄹ)	(바)	(四)	(木)	(学)		エ	ル	ㄹ	바	項	男	協	
4	(オ)	(ヲ)	(ㅁ)	(사)	(五)	(金)	(祭)		オ	ヲ	ㅁ	사	特	女	名	
5	(カ)	(ワ)	(ㅂ)	(아)	(六)	(土)			カ	ワ	ㅂ	아	増	問	宗	
6	(キ)		(ㅅ)	(자)	(七)	(祝)			キ		ㅅ	자	減	主	労	
7	(ク)		(ㅇ)	(차)	(八)	(資)			ク		ㅇ	차	基	副	学	
8	(ケ)		(ㅈ)	(카)	(九)	(社)			ケ		ㅈ	카	調	略	有	
9	(コ)		(ㅊ)	(타)	(十)	(名)			コ		ㅊ	타	優	大	株	
A	(ロ)		(ㅋ)	(파)	(株)	(財)			ロ		ㅋ	파	財	小	社	
B	(ハ)		(ㅌ)	(하)	(有)	(至)			ハ		ㅌ	하	秘	祝	監	
C	(ニ)		(ㅍ)	(주)	(代)	(企)			ニ		ㅍ		印	休	資	
D	(ホ)		(ㅎ)		(呼)	(労)			ホ		ㅎ		上	特	可	
E	(ヘ)		(가)		(協)	(祉)			ヘ		가		中	写	[illegible]	
F	(ト)		(나)		(日)	(監)			ト		나		下	控	[illegible]	

	CJK Squared Japanese								CJK Squared Latin Abbreviations							
	440	441	442	443	444	445	446	447	448	449	44A	44B	44C	44D	44E	44F
0	明治	アパート	ガロン	コルナ	バーレル	ポイント	メートル		pA	Hz	cm^2	ps	kΩ	lm		
1	大正	アルファ	ガンマ	コーポ	ピアストル	ボルト	ヤード		nA	kHz	m^2	ns	MΩ	ln		
2	昭和	アンペア	ギガ	サイクル	ピクル	ホン	ヤール		μA	MHz	km^2	μs	a.m.	log		
3	平成	アール	ギニー	サンチーム	ピコ	ポンド	ユアン		mA	GHz	mm^3	ms	Bq	lx		
4		イニング	キュリー	シリング	ビル	ホール	リットル		kA	THz	cm^3	pV	cc	mb		
5		インチ	ギルダー	センチ	ファラッド	ホーン	リラ		KB	μℓ	m^3	nV	cd	mil		
6		ウォン	キロ	セント	フィート	マイクロ	ルピー		MB	mℓ	km^3	μV	C/kg	mol		
7		ウルシ	キログラム	ダース	ブッシェル	マイル	ルーブル		GB	dℓ	m/s	mV	Co.	PH		
8		エスクード	キロメートル	デシ	フラン	マッハ	レム		cal	kℓ	m/s^2	kV	dB	p.m.		
9		エーカー	キロワット	ドル	ヘクタール	マルク	レントゲン		kcal	fm	Pa	MV	Gy	PPM		
A		オンス	グスーム	トン	ペソ	マンション	ワット		pF	nm	kPa	pW	ha	PR		
B		オントロ	グラム	ナノ	ペニヒ	ミクロン			nF	μm	MPa	nW	HP	sr		
C		オーム	グラムトン	ノット	ヘルツ	ミリ			μF	mm	GPa	μW	in	Sv		
D		カイリ	クルゼイロ	ハイツ	ペンス	ミリバール			μg	cm	rad	mW	KK	Tel		
E		カラット	クローネ	パーセント	ページ	メガ			mg	km	rad/s	kW	KM	Wb		
F	株式会社	カロリー	ケース	パーツ	ベータ	メガトン			kg	mm^2	rad/s^2	MW	kt			

[1] 一部

画	Big5	GB	JIS	KS
	一 A440	一 5027	一 1676	一 7673
1画	丁 A442	丁 2201	丁 3590	丁 7943
	七 A443	七 3863	七 2823	七 8650
2画		卫 4632		
	三 A454	三 4093	三 2716	三 6318
	上 A457	上 4147	上 3069	上 6330
	万 C945	万 4582	万 4392	万 5618
	下 A455	下 4734	下 1828	下 8927
	丈 A456	丈 5341	丈 3070	丈 7759
	丌 C946	丌 5602	丌 D-15	
3画	丏 C94D			
	丆 C94E			
	不 A4A3	不 1827	不 4152	不 6084
	丑 A4A1	丑 1983	丑 1715	丑 8568
	与 C94F	与 5175	与 4531	
	丐 A4A2	丐 5604	丐 4802	
4画		丛 2052		
		东 2211		
		兰 3228		
		业 5021		
	丙 A4FE	丙 1791	丙 4226	丙 6016
	且 A542	且 3950	且 1978	且 8306
	丘 A543	丘 3980	丘 2154	丘 4688
	世 A540	世 4232	世 3204	世 6506
	丕 A541	丕 5607	丕 4803	丕 6164
5画			両 4630	
	丟 A5E1	丢 2210	丟 D-41	
	丞 A5E0	丞 5609	丞 3071	丞 6710
6画		丽 3286		
6画		两 3329	两 D-51	
	更 A7F3	更 2492	更 2525	更 4458
7画	並 A8C3		並 4234	
10画	爾 BAB8	爾 1-2291	爾 2804	爾 7619

2 部

画	Big5	GB	JIS	KS
		丨 5613	丨 D-67	
2画		个 2486	个 4804	
	丫 A458	丫 4930	丫 D-71	
3画	丮 C950		丮 D-74	
		书 4273		
	丰 A4A5	丰 2365	丰 D-76	
	中 A4A4	中 5448	中 3570	中 8173
4画	丱 C963		丱 4805	
6画	串 A6EA	串 2014	串 2290	串 4613
7画	丳 CBB1			

[3] 丶部

画	Big5	GB	JIS	KS
		丶 5628	丶 4806	
2画	丸 A459	丸 4572	丸 2061	丸 9215
3画	丹 A4A6	丹 2104	丹 3516	丹 5101
	之 A4A7	之 5414	之 3923	之 8193
4画	丼 C964		丼 4807	
	主 A544	主 5487	主 2871	主 8111
7画		临 3357		
8画		举 3057		

[4] 丿部

画	Big5	GB	JIS	KS
		丿 5615	丿 4808	
1画	乂 C940		乂 4809	乂 7149
	乃 A444	乃 3643	乃 3921	乃 5012
2画		义 5069		
	久 A45B	久 3035	久 2155	久 4689
	么 A45C	么 3520		
2画	乇 C947	乇 5617	乇 D-120	
4画		乐 3254		
		乌 4654		
	乏 A546	乏 2306	乏 4319	乏 8925
	乎 A547	乎 2685	乎 2435	乎 9126
	乍 A545	乍 5307	乍 3867	乍 6231
5画	乓 A5E3	乓 3750		
	乒 A5E2	乒 3825		
7画	乖 A8C4	乖 2552	乖 4810	乖 4650
	脊 AAD4	脊 7540	脊 2672	脊 9302
9画			乗 3072	
	乘 ADBC	乘 1943	乘 4811	乘 6711
	喬 B3EC	喬 1-3939	喬 2212	喬 4666

[5] 乙部

画	Big5	GB	JIS	KS
	乙 A441	乙 5050	乙 1821	乙 7564
1画	九 A445	九 3037	九 2269	九 4690
	乜 C941	乜 5631	乜 D-166	
2画		飞 2341		
	乞 A45E	乞 3882	乞 2480	乞 4387
	也 A45D	也 5018	也 4473	也 6905
5画				乫 4265
				乭 5244
	乩 A5E4	乩 5632	乩 D-182	
6画		龟 2574		
		乱 3450	乱 4580	
7画				[illegible] 606
				乷 631
	乳 A8C5	乳 4073	乳 3893	乳 746
8画				

部(8画) => 乙部(8画)

	Big5	GB	JIS	KS
62] 進部				
3畫	迂 A8B1	迂 5156	迂 1710	迂 7370
4畫	迗 CD02			
	迖 CD00			
			D-38735	
	迒 CDCC		迒 D-38755	
	迋 CDCD		迋 D-38737	
	迍 CDCF		迍 D-38747	
			辿 7773	
		迟 1957		
		还 2725		
		进 2988		
		连 3312		
		违 4605		
		远 5222		
		运 5243		
		这 5366		
	返 AAF0	返 2321	返 4254	返 5887
	近 AAF1	近 2992	近 2265	近 4846
	迎 AAEF	迎 5113	迎 2362	迎 7142
	迓 CDCE	迓 6934	迓 D-38756	
	迕 CD01	迕 6935	迕 D-38759	
5畫	迣 D0B3		迣 D-38783	
	迠 D0B6		迠 D-38777	
	迡 D0B4			
				迲 4406
			迯 7777	
		迩 6939	迩 3886	
		迳 6941		
	迪 AD70	迪 2147	迪 7776	迪 7872
	迭 ADA1	迭 2192	迭 3719	迭 8287

	Big5	GB	JIS	KS
5畫	迫 ADA2	迫 3840	迫 3987	迫 5862
	述 AD7A	述 4266	述 2950	述 6691
	迢 AD7C	迢 4486	迢 7775	
	迥 AD7E	迥 6936	迥 7774	
	迮 D0B5	迮 6937	迮 D-38801	
	迤 ADA3	迤 6938	迤 D-38785	
	迦 AD7B	迦 6940	迦 1864	迦 4228
	迨 ADA4	迨 6942	迨 D-38791	
6畫	迼 D3F0			
	迾 D3EA		迾 D-38837	
	迵 D3EB		迵 D-38821	
	迿 D3ED			
	迶 D3F1		迶 D-38822	
	迻 D3EE		迻 D-38833	
	迺 B069		迺 7782	
	迴 B06A		迴 7779	
			逈 D-38868	逈 9113
		迹 2803	迹 7781	迹 7873
		选 4901		
		逊 4923		
	迸 B06E	迸 1737	迸 7794	
	迷 B067	迷 3552	迷 4434	迷 5827
	逆 B066	逆 3670	逆 2153	逆 7029
	适 D3EC	适 4242	适 D-38844	适 4633
	送 B065	送 4345	送 3387	送 6574
	逃 B06B	逃 4451	逃 3808	逃 5217
	退 B068	退 4543	退 3464	退 8760
	追 B06C	追 5523	追 3641	追 6558
	逅 B06D	逅 6943	逅 7780	逅 9317
	逄 D3EF	逄 6944	逄 D-38847	

	Big5	GB	JIS	KS
7畫	逤 D7E2			
	逜 D7E5			
	逌 D7E6		逌 D-38872	
			逧 7790	
			逎 7805	
			逓 3694	
		递 2161		
		逦 6946		
	逞 B378	逞 1949	逞 7787	逞 5433
	逗 B372	逗 2226	逗 3164	逗 5272
	逢 B37B	逢 2374	逢 1609	逢 6081
	逛 B37D	逛 2568	逛 D-38893	
	連 B373	連 1-3312	連 4702	連 5407
	逝 B375	逝 4237	逝 3234	逝 6406
	速 B374	速 4357	速 3414	速 6560
	通 B371	通 4508	通 3644	通 8755
	透 B37A	透 4524	透 3809	透 8766
	途 B37E	途 4530	途 3751	途 5218
	造 B379	造 5276	造 3404	造 8067
	這 B36F	這 1-5366	這 3971	這 7847
	逐 B376	逐 5480	逐 3564	逐 8579
	逕 B377	逕 1-6941	逕 7784	逕 4479
	逋 D7E3	逋 6945	逋 7789	逋 8871
	逑 D7E4	逑 6947	逑 7783	逑 4739
	逍 B370	逍 6948	逍 7786	逍 6546
	逖 B37C	逖 6949	逖 7788	
	逡 D7E7	逡 6950	逡 7785	逡 8168
8畫				
			D-38950	
	逴 DC4E		逴 D-38947	
	週 B667		週 2921	週 8146

56

56 Tokens

56

Architecture

Tokens are fast strings. A token is an object that has a unique id, one id per string. The token server guarantees that two identical strings will return the same unique id. Tokens are useful for sending strings within your application, or across applications on the same machine. Tokens are streamable, and will do the right thing — either streaming the unique id if the stream is a local stream, or streaming the string if the stream is across the network, or to the deep freeze.

The system maintains a Token Server that performs the work of token issuance. The application framework maintains a local cache so that second and further token requests for the same string do not incur IPC overhead.

Tokens may be static objects. They have been designed not to go to the Token Server until the first time they are used.

Tokens are most useful where there is a need for extensibility. An example of this is the event name space. In order to support new input devices the event space cannot be limited to a 16 bit bit-mask as is done today, instead events are just strings like "MouseUp", and the space of all strings is allowed. Another example is the text layout classes. Each of the styles in a text layout is a token rather than a const integer so that it will be possible for developer's to add new styles without the naming conflicts inherent in a integer based scheme (i.e. what does number 100 mean?).

There are definitely places where tokens are either the wrong solution, or are overkill. The server architecture used tokens originally, but because there is no mechanism for extending server messages it was decided that a simple integer based scheme would be fine. Sending tokens across the network is another area where it is probably not a good idea, unless of course you want to provide a Network Token Server.

Usage

It is preferable to store a tokens as a **TToken** rather than **TToken***. Typically tokens are passed by reference if they are to be filled in, const reference if they are read only.

A token may be **invalid**, i.e. not have a string. An invalid, or blank, token is useful when you want to copy a token, or use the `operator=` function.

Static tokens are useful. In the following code, the "MouseDown" token is instantiated once, and then used for comparison purposes thereafter.

```
static TToken mouseDown("MouseDown");
TToken eventName;
event->GetEventName(eventName);      // fill in eventName
if (eventName == mouseDown)          // perform comparison with static
    ...
```

56

Making Whoopee

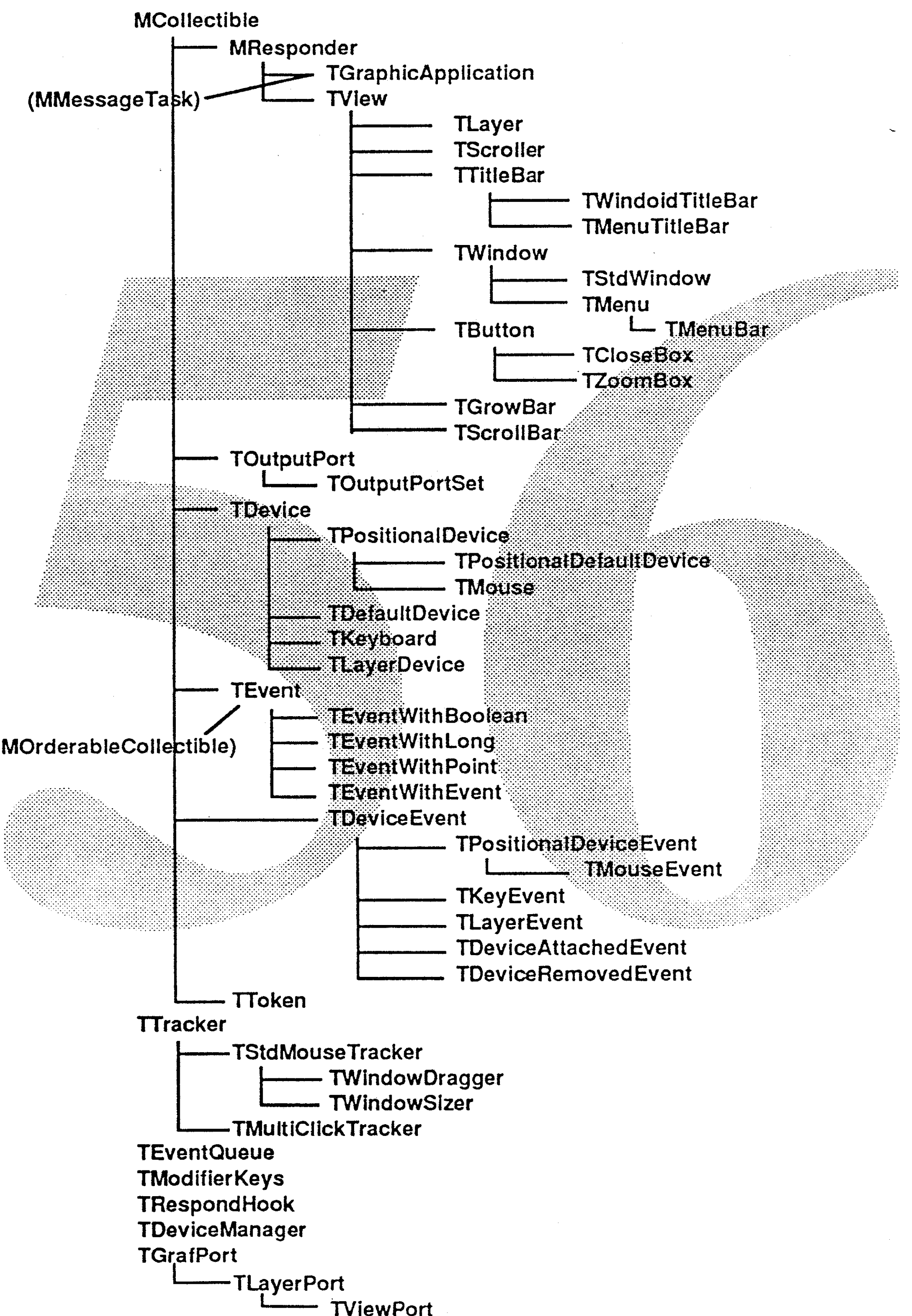
a.k.a.

The Application Framework

56

The Class Hierarchy

The class hierarchy for the application framework is substantial, and looks like this. Notice that most objects descend from MCollectible, making it possible to use them with the collection classes.

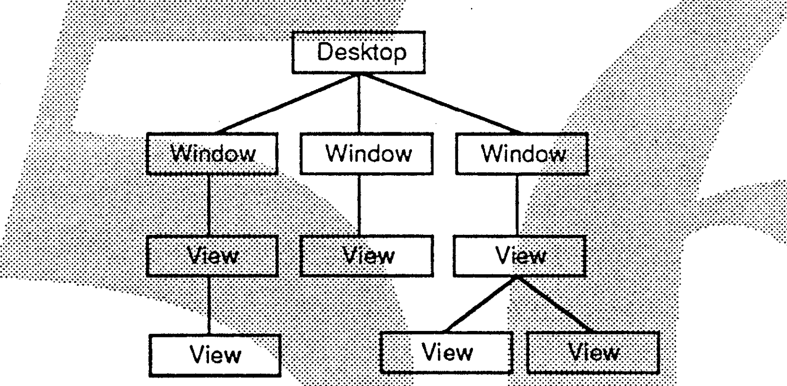


The most important classes are **TGraphicApplication** from which your application must descend, **TView** and its subclasses which allow you to subdivide the screen into logical drawing areas, **TViewPort** which you use to draw into a view, and **TEvent** and **TDeviceEvent** which are created either by external events, or may be created by you to send messages to yourself or other applications.

The Visual Hierarchy

The visual hierarchy describes all the visible objects, or **views**, that the application framework knows about. The visual hierarchy is built around the idea of **enclosures**. Everything that you see on the screen belongs to — is enclosed by — another visual entity. Every enclosure is a subclass of the view class.

At the top of the visual hierarchy is the desktop. The desktop encloses all of the windows in your application, as well as those in all other applications that are running simultaneously. Each window encloses one or more views, and views can enclose other views. This is a typical visual hierarchy. [Note: we should think about making the desktop a view also. That way we can encapsulate layer switching protocol into view methods like RespondPositionally(), rather than hard coding it into some combination of the event and layer servers]



The visual hierarchy is dynamic, changing as your program runs. When you open a new document, a new window is added to the desktop, and views are added to the window into which you draw the document's contents.

All drawing takes place in a **viewport**. Viewports are a synchronization mechanism, allowing multiple threads to draw simultaneously into the same view. Drawing that occurs in response to an update request, e.g. after window damage has occurred, must be drawn into a viewport provided by the application framework. Drawing that occurs at any other time must be into a viewport managed by the drawer.

Visual, or positional, events, work their way down the visual hierarchy, from the desktop to the active window, to the appropriate view. Views can handle positional events because they are **responders**. When you click in a view, the application framework uses the visual hierarchy to determine which view the mouse went down in, and tells it to **respond** to the "MouseDown" event. If the view has a mouse down handler it will respond to the event, if not, each successive enclosing view gets a chance to respond to the event until one some view responds.

The Layer Server

The desktop is managed by the layer server, a system service that subdivides screen real estate into layers. Each layer is owned by an application. Your application may decide to place every window in a separate layer, or place multiple windows in a single layer (a la MultiFinder). The default behavior for Pink will be to place every window in a separate layer and use another mechanism to allow the user to group windows into logical projects or rooms (to use Xerox's term). [The layer server would need to be extended to support grouping.]

The layer server model has been designed to support a multi-processor system — all screen real estate changes must be made through the layer server. Update events that are caused by inter-layer changes are the responsibility of the layer server and are transmitted to an application through the event server, just like other external event such as mouse and keyboard events. Intra-layer view updates are handled by the view system within your application's address space, and local update events are posted to the event queue. You as an application writer will never have to deal with update events from either the layer server or view system directly, instead each view that requires updating will have its `Draw()` method called.

Currently the layer switching protocol is built into the layer server and event server. This may need to change to allow the protocol to be modified at run-time. For example, the Finder may want layer switches to occur on mouse up rather than mouse down. This may be done by providing a desktop class which wraps up layer functionality within the view metaphor.

The Responder Chain

The **responder chain** along with the target specifies which object is to handle an event. If a responder can't, or doesn't want to, respond to an event, it passes the event to its **next responder**. If no object wants to handle the event, the application object, at the top of the responder chain, receives the event, and may either respond to or ignore the event.

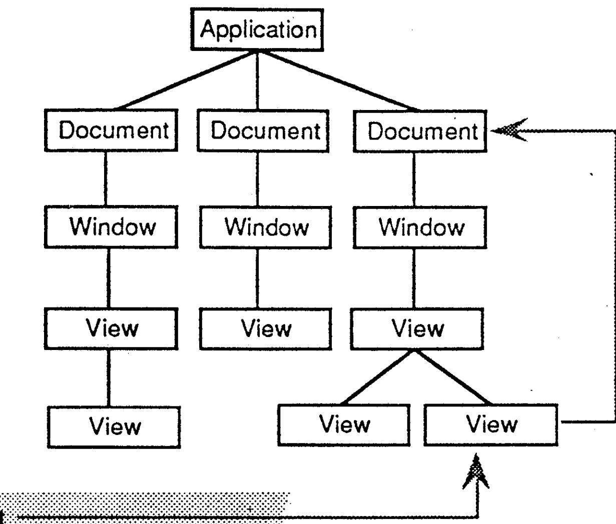
Every object in the responder chain must descend from the responder class. Every responder subclass has a dictionary specifying the events to which it can respond. When a responder is told to `Respond()` to an event, it looks in its dictionary, and the dictionary of all of its superclasses, to see if it understands the event. If it understands the event the handler method is executed, and if it doesn't, it passes the event to its next responder.

The first object to get a chance to respond to an event depends on the type of event. If the event is **non-positional**, it goes to the target. Your application is responsible for setting the target. If the object that the target points to doesn't want to respond to the event, the event is passed to its next responder.

If the event is a **positional event**, the application framework uses the visual hierarchy to pass the event to the view in which the event occurred. If that view does not want to handle the event, every enclosing view is given a chance at the event until either some view responds to the event, or no view responds. If no view responds the application object does not get a chance to respond to the event [should it?].

If the event is **responder specific**, the event is delivered directly to a particular responder regardless of either the target, or if the event is positional, the view the event occured in. The object that is the specific target of the event may decide to either respond to the event, or pass the event to its next responder.

In the following diagram the target points to a view whose next responder is a document. If the view can't handle the event, e.g. a mouse down, it passes the event to the document. If the document can't handle the event it gets passed to the application.



The default behavior for a view is that its next responder is its enclosing view. In the above diagram the next responder for the view pointed to by the target is not its enclosing view, but rather the document. This could have been either because the view didn't want any of the intervening views to get a chance at the event, or because there are so many intervening views that performance is a problem.

Events

An **event** is a just a string, such as "MouseDown" or "Close" or "Zoom". An **device event** is an event as we know it in the Blue system, it has a timestamp, a pointer to the input device that created it, and a position if the event is positional.

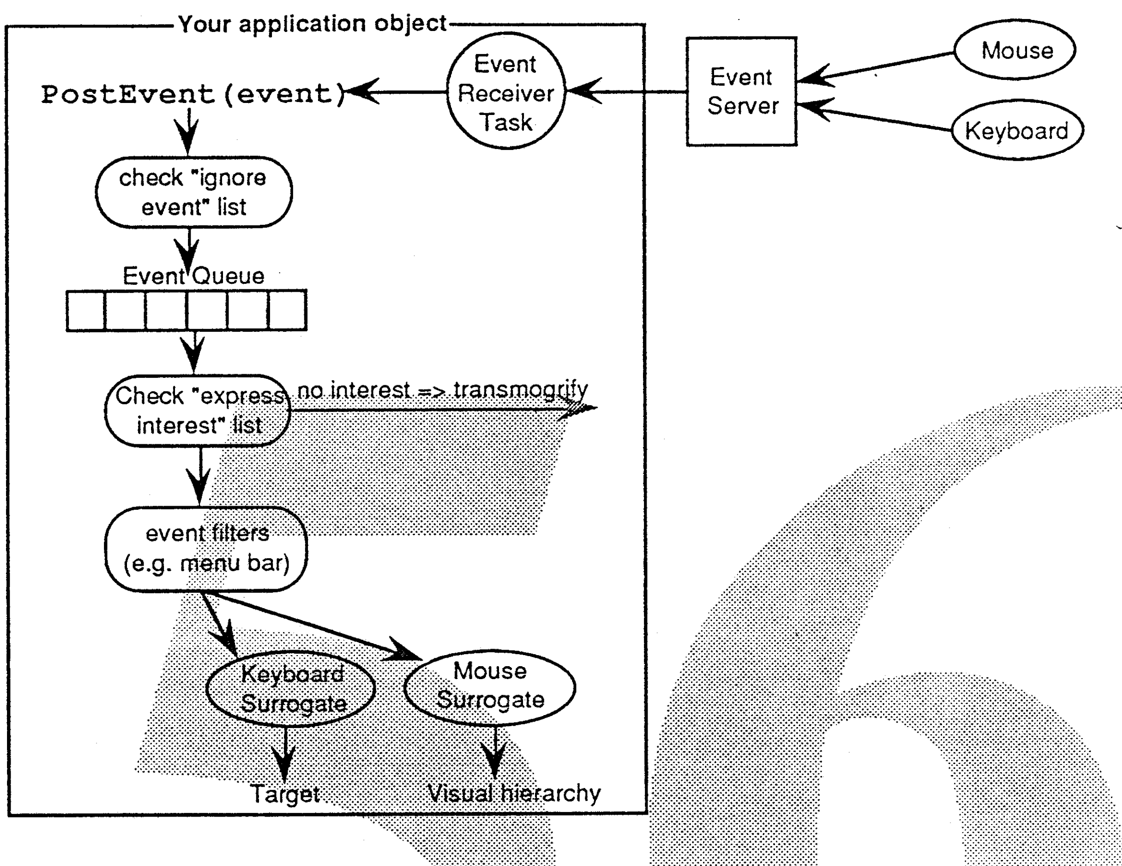
Responders respond to events. Most events that your application will encounter are those created by input devices external to your application, e.g. mouse and keyboard events. When a responder is asked to respond to an event, its `Respond(TEvent *event)` method is called. The responder mechanism dynamically determines the event handler function, and calls it, passing the `event` as a parameter. In order to respond properly the handler will need to cast the `event` into the appropriate type, e.g. a `TMouseEvent*` if the event is a mouse event. This cast can not be done safely, since the handler can only assume that it has been passed an object of the correct type. This can only be solved by providing meta-data calls to the run time system that allow you to check the data type before the cast.

Events are also used to generalize the control of standard interface objects. For example, instead of calling the window's `Close()` method directly, the close box sends the event "Close" to its next responder. No matter how many levels of responder the close box actually is from the window, the window will eventually receive the "Close" event, and close itself. All user interface components will respond to well documented events so that it will be possible to wire together the interface using a NeXT like interface construction kit. When designing your application you should consider where the use of events can help make a component reusable, thereby making your application more extensible.

Another use of events is for synchronization within your own application. If you have multiple threads that want to perform actions on the same object, instead of wrapping that object in semaphores you can post an event to yourself and be guaranteed that the event will be distributed in the application framework's main thread.

Flow of Control

The standard flow of an event from an input device to the correct responder in your program is shown in the following diagram.



Let's review the sequence of events. First the event is created by the user by interacting with a device such as the mouse or keyboard. This event is sent to the **event server** which, in conjuction with the layer server, decides to which application to send the event. The event server send the event to your application's **event receiver task**, which **posts** the event to the **event queue**. Post event first checks the **ignore event list**, and if the event is in the list throws the event away. The application framework blocks until the event queue is non-empty, at which time it pulls the event from the event queue and checks it against the **express interest list**. If no one has expressed interest in the device which created the event, the event is sent back to the device for **transmogrification**. If the event passes the express interest check, any objects that have registered as **event filters** are given first crack at the event. The most common, and typically only, event filter is the application menu bar which looks at every event to check for menu commands. All events that were not used by a filter are sent to the **surrogate** that created the event, allowing that surrogate to distribute the event as it wants. Most surrogates will just use the default logic — **positional** events are sent to the view in which they occured, **non-positional** events use the target, and **responder specific** events are sent to the indicated responder.

Event distribution is guaranteed to be synchronous, i.e. the application framework always distributes events in its main thread.

The application framework gives synchronous control to your application under two other conditions — when you are **tracking**, and when you have requested **idler time**. If your application is tracking an input device, like the mouse, you are given time periodically (at a frequency of your choosing), until the tracking completes. The tracking framework allows for tracking multiple input devices simultaneously, and for distributing events while tracking is occuring. Tracking time is given synchronously rather than asynchronously because it would be very difficult, if not impossible, for you to properly synchronize multiple devices tracking, or to handle eventsasynchronously while tracking. Idler time is given only when your application has been idle for a short amount of time, i.e. no events have been distributed, and

no tracking is occurring.

Event Distribution

As described in the "Flow Of Control" section, above, there are three types of events, **positional**, **non-positional**, and **responder specific** events. The application framework takes care of getting these events to the proper responders — positional events go to view that they occurred in, non-positional events go to the target, and responder specific events go to the indicated responder.

It is possible to **bind** a particular input device directly to a responder by using the `TInputDevice::SetSpecificTarget()` method, such that all events created by that device go directly to that responder. This should only be done in circumstances where the user cannot possibly become confused. A game is one example, where multiple mice or keyboards are view specific. Another example might be the use of a DataGlove, where all input from the glove is directed to the window with the architectural drawing, thereby allowing the user to manipulate the scene even though the window is not front-most.

[Need device binding at global level?]

The primary goal of the event distribution system is to allow new events from new devices to be properly handled. There are two ways this is accomplished. One is by using strings for events, rather than a bit mask. Using strings allows a completely unbounded name space for events. The other way is by routing all events to their surrogates for distribution. New devices might want to distribute their events based on criteria other than the four that are built in — to the target, to the view, responder specific, or binding to a responder.

Graphic Application

Architecture

Every graphical program must create a subclass of TGraphicApplication and create only one instance of this class. The application is the highest level in the responder chain. It is the only responder without a next responder.

The application thread is typically the main thread, and while this is preferred, it is not required. The application object instantiates several supporting objects, including the event queue, tracker and idler queues, source manager, and application menu bar, along with one light-weight thread, the event receiver task.

The motto of the application framework is "don't call me, I'll call you." If you are familiar with MacApp or Think's object-oriented framework you will have little problem adapting to this style of programming. If you are more comfortable with the standard Macintosh style of programming, where you write the main event loop, and control the distribution of events, you may have a little trouble adapting.

The goal of the application framework is that events from new input devices, including devices we can't possibly think of today, will be distributed correctly, as soon as those devices become available, without having to rewrite your application — functionality available on no other computer platform. To do so requires that the event distibution mechanism be controlled by the application framework.

MainEvent Loop

Once started, the application goes into its main event loop, blocking until an event is placed in the event queue. Each event is removed and distributed in timestamp order. After distribution, the event is deleted[1], and the application blocks until another event is placed in the event queue. In addition to distributing events, the main event loop gives time to trackers, and if the application has been idle for a short amount of time, to responders in the idler queue. All three of these functions occur synchronously, in the application's main thread.

Trackers are given periodic time while the user is tracking with an input device. It is possible to have multiple trackers tracking simultaneously if multiple input devices are attached to the system — in that case each tracker is given time synchronously. While tracking, incoming events will be distributed and processed[2]. [Note: the synchronization of events that affect objects that are being tracked

Any responder in your application can request that it be given periodic time when the system has been idle for a small amount of time. The application framework senses when the system has blocked for a small amount of time—i.e., no events have been distributed, and no trackers have been given time—and looks in the idler queue to see if there are any idlers[3].

---

1. To keep a reference to an event you must call the event's `Clone()` method, you must not keep a ptr to it since the application framework deletes each event after it has been distributed. See the chapter "???" for more information.

2. This works now, but we have yet to see whether this is something that is both useful and understandable. It may be disconcerting to have certain events distributed during tracking, e.g. closing a window with a keyboard command at the same time that you are dragging the window.

3. It is possible to supply behavior equivalent to the idler queue by creating a thread, but threads require synchronization, and it is often useful to be able to get time synchronously, when you can be sure that no events are being distributed. Consider blinking the text cursor—if the blinking is handled as a separate thread, it is necessary to synchronize the blinking with incoming keystrokes, using semaphores,

Application State

When an application is launched it may or may not require an interface. For example, the Finder might launch an application just to perform text retrieval, to make a print request , or to grab data from a link. In these cases there is no need for an interface.

[I don't know what's going to happen here yet, but I know it should be something ...]

Priorities

When an application is started, it is given a preset priority by the application framework. This priority has been carefully tuned to provide optimal system performance, particularly in regards to user tasks such as handling mouse and keystroke events, and performing device tracking. When an application is brought to the front, the priority of all threads are changed such that the front-most application's user interface has the highest priority. All other threads in an application are assigned to one of a number of pre-specified categories, such as **animation**, **cpu intensive**, and **server** categories. Refer to the "Opus Wrappers" documentation for a description of these categories.

Class Diagram

See first page of this chapter.

Usage

TGraphicApplication must be overridden to implement an application with the Pink Toolbox Libraries. An application usually has one or more documents associated with it, and while the CHER classes allow you to create documents, there is no document support in the application object yet (suggesting perhaps that there will be in the future).

The application is the root of the responder chain. Based on the type of event — non-positional, positional, or responder specific — each event will be sent to some responder in the responder chain. If no responder handles the event it will end up at the application object. If the application can't handle the event, the message is ignored.

Writing the main program

In your C++ `main()` function, you must first instantiate an application object. After instantiation you tell the application to start running by calling its `Start()` (a method of MMessageTask).

```
main()
{
    // Create application object, and start it running

    TMyApplication app();           // see header file for any parameters
    app.Start();                    // method of MMessageTask()
```

so that, for example, the cursor positioning code doesn't interfere with the character insertion code.

```
    // If we get here then the user has quit the app.

    return 0;                      // for C runtime clean-up
}
```

Your application object

Your application object will be a subclass of TGraphicApplication[4], and will look something like this:

```
TMyApplication :: TGraphicApplication {

    protected:
        virtual void    Main(TMemory &);

    public:
        TMyApplication(...);
        virtual ~TMyApplication();

        virtual void        AboutToQuit();
};
```

`void TMyApplication::Main(TMemory &)`

This is a method of MMessageTask, and is called by the application's `Start()` method (**Note: you must never override `Start()`**). You should override `Main()` if you want to perform actions after the constructor, but before your application enters its main event loop—opening windows, starting background tasks, or installing responders in the idler queue. After performing your actions you must call the inherited `Main()`.

`void TMyApplication::AboutToQuit()`

Upon exiting from the main event loop, the application framework calls this method. You should perform any necessary clean up at this time.

Methods You May Want To Override

`void TGraphicApplication::BecomeActiveApp()`

If you want to be notified when your application becomes active, or front-most, you should override this method. You must call the inherited `BecomeActiveApp()`.

`void TGraphicApplication::ResignActiveApp()`

If you want to be notified when your application becomes inactive, or no longer front-most, you should override this method. You must call the inherited `ResignActiveApp()`.

---

4. Applications that do not need a user interface, e.g. pure servers, may descend from TApplication instead.

Methods You May Want To Call

`void TGraphicApplication::RegisterEventFilter(MEventFilter *);`

Objects, like the menu bar, that want to get first crack at each event before it is distributed, may register as an event filter. After each event is pulled off the event queue, but before it is distributed, all registered MEventFilter's are passed the event by calling `MEventFilter::CheckForCommand()`. The menu bar gets first crack, and all other MEventFilter's are given a chance in the order in which they were registered. If any of the calls to `MEventFilter::CheckForCommand()` return `TRUE`, no other MEventFilter's are checked and the event is not distributed.

`void TGraphicApplication::UnregisterEventFilter(MEventFilter *);`

The opposite of `RegisterEventFilter()`.

`static TGraphicApplication* TGraphicApplication::GetRunningApplication() const;`

This method allows any object in your application to get a pointer to the application object.

`static TSourceManager* TGraphicApplication::GetSourceManager() const;`

This static member function allows any object in your application to get a pointer to the source manager object.

`static TEventQueue* TGraphicApplication::GetEventQueue() const;`

This static member function allows any object in your application to get a pointer to the event queue object.

`static TWindowIterator* TGraphicApplication::GetWindowList() const;`

This static member function allows any object in your application to get an iterator to the window list for all windows that reside in separate layers. This is a read only iterator making it impossible for you to manipulate the window list — you must use the view system to manipulate the window list.

This iterator is protected by a semaphore so that the window list will not be changed while you iterate. You must hold the iterator for as short a time as possible since all view changes are blocked until you delete the iterator. Additionally, only one iterator is available at a time so I don't have to go through semaphore machinations, you will receive a `NIL` iterator if one is already outstanding.

Event, Tracker and Idler Queues

Architecture

Events may be placed in the **event queue** from either an external source, such as an input device, or any object within your application. External events come via the event server, and are posted to the event queue by the event receiver task which calls `TEventQueue::PostEvent()`. If your application wishes to post an event to itself, it may use either `TEventQueue::PostEvent()`, or `TEventQueue::PostEventTo()`. If you wish to post an event to another application you may use `TEventQueue::PostEventToApp()`.

All calls to the event queue are protected by semaphores, so they can be called safely from any thread. The event queue is of unlimited size, meaning that all events are guaranteed to be delivered, unless the application quits with events still in the queue. Events are either positional, non-positional, or responder specific, and are delivered in timestamp order.

It is possible to ask the event queue to filter out unwanted events. Using the methods `IgnoreEvent()` and `AcceptEvent()` it is possible to request that particular events not be posted to the event queue. The application framework default is that the events "KeyUp", "ModKeyUp" and "ModKeyDown" are ignored. If your application wants to get any of these three events you will need to call `AcceptEvent()`.

There are two other queues in the application framework, the **tracker** and the **idler queues**. Tracker objects are placed on the tracker queue whenever the user is using an input device to track something, to drag the window for example. Objects in the tracker queue get periodic time, at least until we implement "MouseMoved" events. Every time one of the tracker alarms goes off, the main event loop wakes up and calls the tracker method `TTracker::TrackContinue()`.

Idlers installed in the idler queue are much like trackers in that they receive periodic time, the difference being that they only receive it if there has been a lull in the system, i.e. no events have been distributed and no trackers have been given time for a pre-specified amount of time. Idlers are useful for things like the blinking text cursor, but can not be used to receive guaranteed periodic time.

Events, trackers and idlers are all given time synchronously, from your application's main thread. This means that synchronization is unnecessary if you only perform actions based on an incoming event, at track continue time, or when an idler gets time.

Class Diagram

See first page of this chapter.

Usage

Event Queue

The event queue is instantiated by the application framework, you never create one. You may get a pointer to the event queue by calling `TGraphicApplication::GetEventQueue()`, a static member function of TGraphicApplication. Events that are received from the event server are automatically placed in the event queue by the event receiver task which calls `TEventQueue::PostEvent()`. All event queue calls are protected by semaphores, so you can call them at any time, from any task.

You may post an event to yourself using either `PostEvent()` or `PostEventTo()`, or to another application using `PostEventToApp()`. There are several calls allowing you to peek at and get the next event.

There are two event ownership rules. Events that you remove from the event queue yourself, using any of the various peek and get event calls must be deleted by you. Events that are distributed to you by the event system are owned by the event system. You must never delete them, and you must never keep a pointer or other reference to them — if you need the event, you `Clone()` it.

Tracker Queue

To place a tracker in the tracker queue you instantiate a tracker, and then call its `StartTracker()` method. The tracker queue recognizes when a tracker has completed tracking, by calling `TTracker::GetPhase()` and comparing the result to the constant `TTrackerPhase::kTrackCompleted`. When the tracker has completed it is automatically removed from the tracker queue (there is no way to stop a tracker prior to its completion).

Idler Queue

To place a responder in the idler queue, you call its `InstallAsIdler()` method. There are two parameters to `InstallAsIdler()` — the frequency of idler time, and the message to be sent to the responder when idler time is given. This allows the same idler to be in the idler queue multiple times, once for each message. Because idlers only receive time when the system is idle, they may not receive idle messages at the requested frequency. For example, if an idler has been installed to receive the message "second" once a second, and the system is busy for five seconds, only one "second" message will be received, after the system has been idle for a small amount of time.

Methods you may want to override

You cannot override the event queue, and the tracker and idler queues are inaccessible, so you're out of luck here.

Methods you may want to call

Event Queue

You may want to call a lot of the methods in TEventQueue. Refer to the 411 documentation for details.

Tracker Queue and Idler Queue

Neither class is accessible. You add a tracker to the tracker queue by calling its `StartTracking()` method. You add a responder to the idler queue by calling its `InstallAsIdler()` method.

Events

Architecture

Events are strings — the mouse creates "MouseUp", "MouseDown" and "MouseMoved" events, the keyboard creates "KeyUp", "KeyDown", "AutoKey", "ModKeyUp" and "ModKeyDown" events. Strings are used to allow for an unbounded name space, since new input devices may require more event names than we can possibly reserve in a bitmask.

Using strings for events has two useful side affects. One, events can come from user scripts as well as input devices, since most responders respond to events regardless of which device created it. Two, the user interface components use events to communicate with one another, allowing for runtime wiring of the interface, and the building of reusable interface components

There are three kinds of events, **positional**, **non-positional**, and **responder specific**. Positional events use the view system to determine which view is under the event, non-positional events are sent to the target, and responder specific events are sent directly to a indicated responder.

Events as we know them in the Blue world are a subclass of event called device events, that include the normal event string along with information about the device that created the event.

There are two event ownership rules. Events that you remove from the event queue yourself, using any of the various get event calls must be deleted by you. Events that are distributed to you by the event system are owned by the event system — you must never delete them, and you must never keep a pointer or other reference to them — if you need to keep a reference to the event, you must `Clone()` it.

Class Diagram

See first page of this chapter.

Usage

Important: See the event "ownership" rules in the architecture section above.

You may create your own events at any time, and either send them directly to the responder chain by calling any responder's `Respond()` method, or post them your application's event queue using `PostEvent()` or `PostEventTo()`, or post them to another application using `PostEventToApp()`.

Posting an event to yourself is a useful method of synchronization. Any thread can post an event, and then the application framework will distribute the events synchronously, in the framework's main thread.

There are several TEvent subclasses that you may find useful — TEventWithBoolean, TEventWithLong, TEventWithPoint, TEventWithEvent. These allow you to package an event with another value that you can then send. For example, the event "MoveTo" requires a point, so you would create a TEventWithPoint, and "DragSelf" requires a mouse event, so you would create a TEventWithEvent.

Methods you may want to override

You will rarely subclass from the event classes unless you are writing your own source, e.g. a driver for a new input device. In that case you should subclass from whatever event most closely resembles the event that your device produces. For example, if you have a new type of keyboard, that adds pressure information, then you might subclass from TKeyEvent. If you have a DataGlove, you may either subclass from TMouseEvent because your events are also positional, or you may instead want to subclass directly from TPositionalEvent (you may in fact want to create an abstract class T3DPositionalEvent that subclasses from TPositionalEvent).

Methods you may want to call

This is completely event specific. See the 411 documentation for details.

Devices and the Device Manager

Architecture

The design of devices and the device manager allows new and unusual input devices, both physical and logical, to be easily designed, automatically attached, and immediately usable by every application. See the paper "A Design for Supporting New Input Devices" by Michael Chen and Frank Leahy for background information.

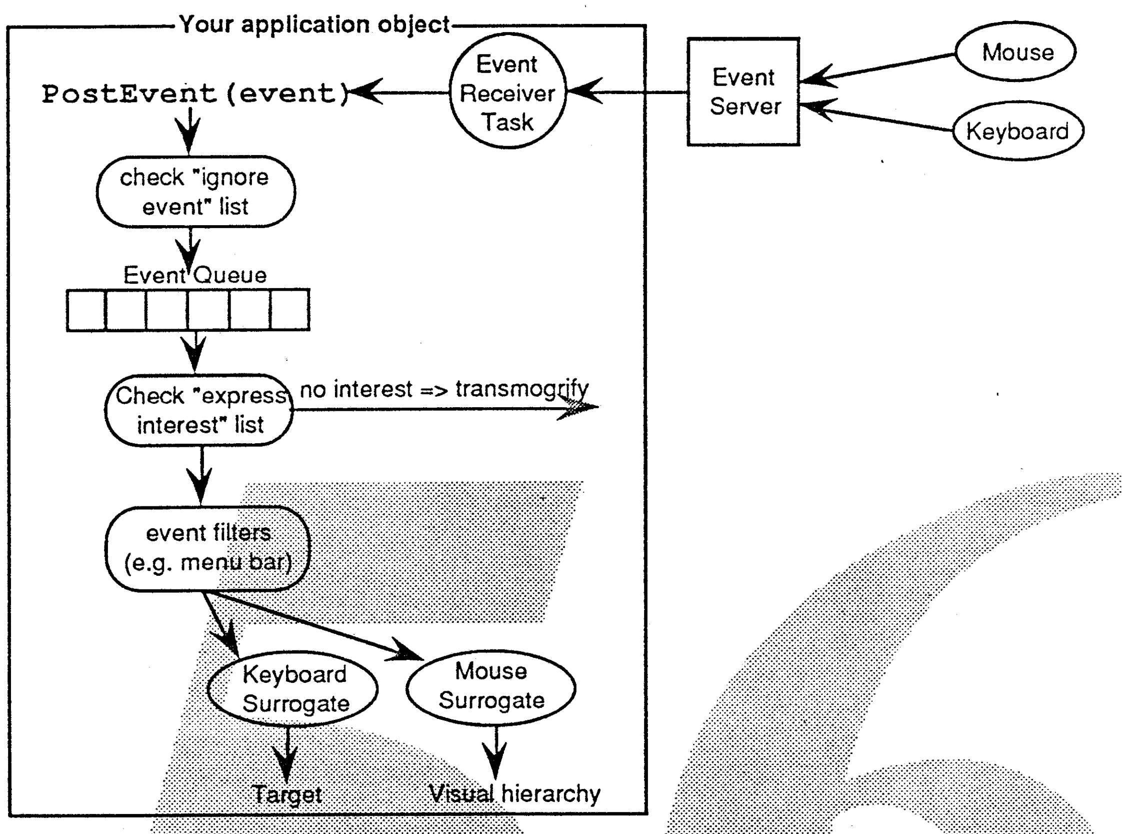
The **device manager** contains references to all devices that are attached to the system. You can query the device manager to see what devices are attached, and get a pointer to a particular device based on its class and **name**.

The device manager keeps **an express interest list**. Before an event is distributed, the express interest list is checked, and if no one has expressed interest in the device that produced the event, the event is sent back for transmogrification before it is ever distributed. Doing this speeds up the distribution of events for which no one has expressed interest, since if no one has expressed interest then they would be dropped on the floor anyway. The application framework expresses interest in the following device classes at startup time — mouse, keyboard, and layer device — this means that your application will get mouse and keyboard events, and respond to updates properly.

The rationale for having express interest list is so that applications can be easily extended to use new input devices. Consider an application that has been designed to accept new tools, such as a paint program that looks in a special folder for externally written tools. New tools written to use new input devices, such as a pressure sensitive tablet, should be able to get events from that new device even though the original paint program knows nothing about it. When the tool is selected, it expresses interest in the new device by calling `TDeviceManager::ExpressInterest(newDeviceClassName)`, whereby the application will be distribute the events untransmogrified. When the tool is deselected it calls `TDeviceManager::RemoveInterest(newDeviceClassName)`, and the new device's events revert to being transmogrified rather than distributed. By expressing and removing interest properly tools that do not understand the new device's events won't be flooded with events from the new device, events that they would ignore anyway.

There are two types of devices — **positional and non-positional**. Every input device that is physically or logically attached to the system has a device object that gets instantiated either when the application is loaded, or, if the device is attached later, when the device is attached to the system. This device object is a **surrogate** that allows the application to access the device from within the application address space. Typically only the tracker framework will have cause to call device methods, but if you need information from a device at other than event time, you can query it directly after getting its surrogate from the device manager.

The following diagram, taken from a previous section, shows the use of surrogate devices. The surrogates are given the events created by their "real" devices, and they distribute them. The default logic is that positional events use the view hierarchy, non-positional events use the target, and responder specific events go to the indicated responder. It is impossible to predict what types of new input devices and events will appear, and how they should be distributed in an application, we have designed the event distribution mechanism to let the surrogate device decide how to distribute its events.



Each device has two pieces of information associated with it — its class and its name. The class is used to categorize devices, e.g. mouse, keyboard, pressure tablet. The name field is used to discriminate between devices when more than one of the same class is attached simultaneously.

Every device has a **modifier keys keyboard** associated with it. When devices events are synthesized the modifier keys passed with the event are those from its modifier keys keyboard. This discrimination is important when multiple keyboards are attached.

Class Diagram

See first page of this chapter.

Usage

Device Manager

Functions provided by the **device manager** are — attaching and detaching devices, setting the system target, expressing and removing interest in a device class.

The application framework handles attaching and detaching devices and setting the system target, you should have no reason to intercept or override these functions.

The application framework expresses interest in the two ubiquitous devices — the mouse and the keyboard — so you don't have to. If you want to get events from other devices you have to specifically express interest in those devices. Any object in your application may call `ExpressInterest()` or

RemoveInterest() to express or remove interest in a particular device, so that your application does/does not receive it's events. If your application does not express interest in a particular device, all of its events will be transmogrified before your application receives them. If the device can't transmogrify its events the events will not be distributed; this is done so you don't get flooded with events from new devices that your application doesn't support.

Other than expressing interest, you should never have to query the device manager directly. Whenever an event needs to be distributed the application framework handles all calls to the device manager. If you are writing a program that wants to track the mouse at all times, even when the mouse is up (e.g. Jack Palevich's program Neko), you could find the primary mouse by calling

```
TGraphicApplication::GetDeviceManager()->IsAttached(TToken("Mouse"),
TToken("Primary"));
```

Devices

You will rarely have cause to query any of the devices directly. All device interaction at event distribution time is handled by the application framework, and communication with the device needed during tracking (a call to device->UpdateEvent()) is handled by the tracking framework.

Methods You May Want To Override

Device Manager

None, since you can't get the application framework to use a subclassed device manager, you can't override anything.

Devices

The following methods are described for people writing surrogate device objects for a new physical or logical device. It isn't possible to change the behavior of existing devices.

void TDevice::Distribute(Event *)

If you want to distribute the events from your device in a manner different from the standard positional and non-positional distribution, you should override this method.

The standard **non-positional** distribution method is as follows:

1. If the event is NIL, don't distribute anything.
2. Call GetBoundTracker() to see if the device is currently tracking. If it is tracking, call theTracker->HandleEvent(event) to give the tracker the event. This allows the tracker to get the event immediately, rather than having to wait until the next time the tracker is scheduled.
3. If the device is not tracking, see if this is a responder specific event by calling GetDistributeEventTo(). If a specific responder has been defined distribute the event to it.
4. If there is no specific responder, see if there is a specific target for the device by calling GetSpecificTarget(). If there is a specific target distribute the event to it.
5. If there is no specific target, get the standard target by calling GetStandardTarget(). GetStandardTarget() uses the system target, getting it by calling TDeviceManager::GetTarget().
6. Once a target has been determined, call target->Respond(event, TRUE). This call returns a Boolean indicating whether anyone responded to the event.

7. If the event is not responded to by the target, call `TransmogrifyEvent(event)`, asking the device to transmogrify the event. If the event is transmogrified call `target->Respond(transmogrifiedEvent, TRUE)`. Keep transmogrifying the event until either the event cannot be transmogrified, or until someone responds.

The standard **positional** distribution method is the same as for non-positional events, except for steps 4 and 5. In step 4, `GetStandardTarget()` uses the layer id in the positional event to determine which layer's view hierarchy to begin traversing. If there is no layer id, it uses the window list to determine which window the event was under. In step 5, `target->RespondPositionally(event, TRUE)` is called instead of `Respond(event, TRUE)`.

void TDevice::UpdateEvent(TEvent &)

This only needs to be overridden if the device produces trackable events. For example, the keyboard does not produce trackable events, so it does not need to override this method.

Given a reference to its event, the device should do whatever is necessary to update the event. The data that needs to be updated is device dependent, for example the mouse updates the event's cursor location field.

An interesting side-affect of having an `UpdateEvent()` call, rather than allowing trackers to query the device directly, is that it is possible to track non-positional devices. I was able to very easily create example where the arrow keys on the keyboard controlled the mouse. This was done by having Control-DownArrow create a "MouseDown" event and Control-UpArrow create a "MouseUp" event. In between the two events, the tracking framework constantly called the keyboard's `UpdateEvent()` method. I overrode that method to see if the event name was "MouseDown", and if so, updated the mouse position depending on the state of the arrow keys — moving left/right/up/down accordingly.

TGPoint TPositionalDevice::GetCursorLocation();

Override this only if the device is a subclass of TPositionalDevice. Return the current value of the device's cursor location in global coordinates.

TEvent* TDevice::TransmogrifyEvent(TEvent *event);

Attempt to transmogrify `event`. If it is not transmogrified, delete the event, and return `NIL`. If the event can be transmogrified, perform the transmogification, either in place, or by creating a new event. Call `TEvent::SetDeviceClassIsActingLike()` to set the correct device class for the newly transmogrified event.

Return the transmogrified event as the return value. This allows you to either modify the original event, or to create a new event to be returned.

void TDevice::AppDidActivate()

After the application becomes the front app, it calls this method for every device in the device manager. Override this method if your device needs to know that the application was activated. You do not need to call the inherited method.

void TDevice::AppDidDeactive()

After the application resigns as the front app, it calls this method for every device in the device manager. Override this method if your device needs to know that the application was deactivated. You do not need to call the inherited method.

`Boolean TDevice::IsUserVisible()`

Some day, some how, some one, will write a desk accessory to display the names of the input devices attached to the system, and let you change them. Obviously only those devices that are user visible should be displayed.

Methods You May Want To Call

Device Manager

`TDevice* TDeviceManager::IsAttached(const TToken &deviceClass, const TToken &deviceName) const;`

See if there is a device with class `deviceClass` and name `deviceName`. Return it if there is, return `NIL` if there isn't.

`TDevice* TDeviceManager::IsAttached(const TDevice *device) const;`

See if `device` is still attached. Return it if it is, return `NIL` if it isn't.

`Boolean TDeviceManager::AnyAttached(const TToken &deviceClass) const;`

See is there are any devices of class `deviceClass` attached.

`Boolean TDeviceManager::AnyAttached(const TProperties &withThisPropertyList) const;`

See if there are any devices that satisfy all of the properties in the property sheet `withThisPropertyList`. *** Currently unimplemented.

`void TDeviceManager::GetAllAttached(const TToken &deviceClass, TSet &devices) const;`

Return in `devices` all devices that match the class `deviceClass`.

`void TDeviceManager::GetAllAttached(const TProperties &withThisPropertyList, TSet &devices) const;`

Return in `devices` all devices that satisfy all of the properties in the property sheet `withThisPropertyList`. *** Currently unimplemented.

Devices

See the individual device for methods you may be interested in.

Writing New Device Objects

The simplest device is TLayerDevice, the logical device used by the layer server to distribute layer update events. You should look at this code first, and then the mouse and keyboard code for more information. Trust me, it's easy.

There are three pieces of code that need to be written before a new physical input device can be attached to an application. One, the ISR, or Interrupt Service Routine, is a small stub of C or assembly code that passes information to the Access Manager. Two, the AccessManager is a C++ object that does not run at interrupt time, but when combined with the ISR is equivalent to a device driver on the Macintosh. Writing an ISR and Access Manager is completely out of my league, and personally I'd be appalled if I ever had to get anywhere near that stuff. See the Opus Spec for writing such things.

The third piece of code, the device surrogate object, which descends from TInputDevice, exists in the application's address space, and insulates the application framework from having to talk to the ISR or Access Manager directly. This device surrogate is the only piece needed for logical device such as the layer device, or the Finder device. To write the device object, look at the TDevice subclasses and decide which of the devices the device you want to write is most like. Subclass from that. Override at least `Distribute()`, `UpdateEvent()`, `IsUserVisible()`, and `operator>>=` and `operator<<=`, and if it is a positional event, `GetCursorLocation()`.

Responders

Architecture

MResponder is an abstract class that implements the ability to receive and handle events. Any object in your application that responds to events from either within or without your application should descend from MResponder. You generally won't need to subclass from MResponder directly because most of the classes that you will want to subclass from are already subclasses of MResponder.

Caveat emptor. The responder architecture assumes that you are using it from the main thread of the application framework, and is not protected by semaphores to stop you from doing something stupid. Of course it is possible to call `Respond()` or `RespondPositionally()` from a thread other than the application framework thread, and in fact this may be a very useful thing to do, but you must be absolutely sure that the handler methods for an object can not be called by another thread simultaneously, or that you have protected it properly. One way to get around this problem is to post events to yourself, and since event distribution always occurs in the application's main thread, you will be safe.

MResponder has capabilities similar to the MServer class, and once upon a time there was talk of combining the two functions. It was decided though that a server and a responder really do have separate functionality. A responder is designed to respond to either synchronous requests from objects within your application, or asynchronous requests from objects outside of your application that come through a synchronizing bottleneck — the event queue. Responders cannot reply to messages they receive from events, whereas servers can reply. A server must handle the request itself, whereas a responder can pass an event to another responder for handling. Another important feature of a responder is that no synchronization is needed to call a responder's Respond() method — the system guarantee's that it will only call Respond() from the application framework's main thread. And finally, the responder handler lookup overhead was deemed unacceptable for servers.

A responder can either respond to an event or it can pass it to its next responder. A responder decides whether it wants to respond to a particular event by looking in one of its four dictionaries and attempting to match the event name with a corresponding entry in one of the dictionaries. The four dictionaries are the instance script dictionary, the class script dictionary, the instance response dictionary, and the class response dictionary; and they are searched in that order. When a responder's `Respond()` method is called, the event that is passed as the parameter to `Respond()` is looked up in the four dictionaries, and if a match is found the value in the dictionary is either a script to be executed, or a pointer to a handler method to be called.

Because of C++ compile time limitations, every handler function must be of the same type. Every handler takes a `TEvent*` as a parameter. Inside the handler function you must cast the `TEvent*` parameter to whatever object you "know" it is, e.g. to a `TMouseEvent*` in the mouse down handler. Unfortunately without run-time type checking it is impossible to be sure that the stimulus is actually the object that you think it is, and so the type cast is unsafe.

Class Diagram

See first page of this chapter.

Usage

There are **five macros** that you must use when defining a new responder subclass. In your class declaration must include the `ResponderDeclarationsMacro()` like this:

```
class TMyClassName : public TMyClassesParentsName {

    ResponderDeclarationsMacro()

    public:
        ...    // your stuff here

    protected:
        ...    // more of your stuff here
    private:
        ...    // even more of your stuff here
};
```

This macro declares all of the dictionaries that your class needs, as well as some other useful things various methods and static variables. Because the macro has both `private` and `protected` declarations, you should be sure to always use a `public`, `protected` or `private` declaration after using the macro.

In your code you must use at least three of the other four macros. The `ResponderDefinitionsMacro()` is code needed by the `ResponderDeclarationsMacro()`. The `ResponderConstructoBeginMacro()` and `ResponderConstructorEndMacro()` must always be used inside of your class's constructor. Here is a sample of how the simplest responder subclass would look.

```
ResponderDefinitionsMacro(TMySubClassName, TMyClassesParentsName,
                          ExecuteMyResponse(stimulus))

TMySubClassName::TMySubClassName(...)
 :   TMySubClassesParentsName(...)
{
    ResponderConstructorBeginMacro()
    ResponderConstructorEndMacro()
}
```

The three paramters to the `ResponderDefinitionsMacro()` are

- your class's name
- your superclass's name — if you are using multiple inheritance then this must be the class that descends from MResponder
- ExecuteMyResponse(stimulus)   // this must be typed exactly like this, capitals and all

The last macro, `RegisterResponseMacro()` is used to associate handlers with event messages. This macro must be used in between the `ResponderConstructorBeginMacro()` and `ResponderConstructorEndMacro()` macros. To associate three different handlers with incoming events, you would write the following:

```
ResponderConstructorBeginMacro()
    RegisterResponseMacro(TWindow, "MouseDown", HandleMouseDown)
    RegisterResponseMacro(TWindow, "Close", HandleClose)
    RegisterResponseMacro(TWindow, "Zoom", HandleZoom)
```

```
ResponderConstructorEndMacro()
```

The three parameters to `RegisterResponseMacro()` are

- your new class's name
- the string you want to respond to
- the name of the handler function

The handler function must be a `MResponderResponseFn` which is defined as

```
typedef Boolean (MResponder::* MResponderResponseFn)(TEvent *);
```

meaning your handler declarations would look like

```
Boolean TMySubClassName::HandleMouseDown(TEvent *event);
Boolean TMySubClassName::HandleClose(TEvent *event);
Boolean TMySubClassName::HandleZoom(TEvent *event);
```

When writing a handler function, you must remember to first cast the `TEvent*` parameter to whatever type you "know" it really is. For example, in the `MouseDown` handler the event is cast to a `TMouseEvent*`. In the window's `Close` handler the event is cast to a `TEventWithEvent*` because the original event, a mouse down, gets combined with an event, `"Close"`, into a `TEventWithEvent*`. Unfortunately, until and unless we get runtime support for checking types, the type cast is unsafe, you can only hope (pray?) that the `TEvent*` parameter is really what you think it is.

Methods You May Want To Override

There are six methods that you may want to override. Because all of these methods are target related, they are described in the "Target" chapter.

`void MResponder::WantToBecomeTarget()`

`void MResponder::WillingToResignTarget()`

`void MResponder::BecomeTarget()`

`void MResponder::ResignTarget()`

`void MResponder::DeactivateTarget()`

`void MResponder::ActivateTarget()`

Methods You May Want To Call

See the Usage section for a description of the four responder macros that you must use.

```
void MResponder::IsEnabled()
```

Call this to find out if the responder is enabled. If it is disabled it will not respond to any events, nor will it pass events on to its next responder.

```
void MResponder::SetResponderName(const TToken &)
```

It is possible for you to name any or all of your responders. Once named you can look them up by name and class. See `MResponder::GetResponderFromClassAndName()`.

```
static MResponder* MResponder::GetResponderFromClassAndName(const TToken
&className, const TToken &instanceName)
```

The system maintains a dictionary of all named responders. Call this static member function of MResponder to look for a responder of class `className` and name `instanceName`.

The RespondHook

There is a hook in the responder mechanism that allows you to be notified every time `Respond()` is called for *every* responder. Your hook must subclass from TRespondHook. You will need to implement the `Hook()` method. There may be more than one hook, and each hook will get called, one after the other. Of course lots of hooks could affect system performance. To add your hook call the static member function `MResponder::AddRespondHook()`, and to remove it call `MResponder::RemoveRespondHook()`.

Target

Architecture

Every **layer** in your application has its own **layer target**. When a layer switch occurs, the layer manager sets the **system target** to be equal to the new front layer's target. Before the layer switch occurs, the system target is told to **deactivate**, and the target is temporarily set to the application. After the layer switch has occured, the newly front most layer's target is told to **activate**, and the system target is set equal to it.

The target is set by calling `SetTarget()` on any view in the layer (it's a view method). If the layer is front most, then the system target is changed. If the layer is not front most then the target is stored in the layer, and when it becomes front most it changes the system target appropriately. Of course if a layer is not the front layer its target should not be changed since target changes are meant to follow the user's focus which should only occur as the result of user actions, like clicking the mouse in a view, or typing a key.

The target is automatically changed every time the user creates a target changing event. The most common target changing event is a "MouseDown" event. Whenever the user clicks in a view, the view method `WantToBecomeTarget()` is called. If `WantToBecomeTarget()` returns `TRUE`, then `SetTarget()` is called on the view in which the mouse down occured. See the Usage section "Setting and Getting a Layer's Target", below for more detail about the six methods that get called whenever a target change occurs.

When a window first appears it is your responsibility to set its target. You could set it to the window, or content view, or if a blinking cursor is visible to the view with the cursor.

Usage

Setting and Getting the System Target

The device manager keeps track of the system target, and you can get it by calling `TGraphicApplication::GetDeviceManager()->GetSystemTarget()`. You never explicitly set the system target, the layer manager sets it for you, using the target of the front-most layer. You can only set the target of a layer by calling `SetTarget()` on any view in that layer. The view system will ensure that the system target stays in synch with the layer target.

Whenever a layer switch occurs the following actions take place:

- System target's `DeactivateTarget()` method is called.
- System target is changed by the view system.
- New system target's `ActivateTarget()` method is called.

You might use this to perform inactive hiliting on text or graphic selections. For example instead of hiliting selected text you could put an outline around it to show that it is selected but not active.

Setting and Getting a Layer's Target

Each layer keeps track of its own target. You can get it by calling `GetTarget()` on any view in the layer, and set it by calling `SetTarget()` on any view in the layer.

The view system changes the target whenever a mouse down (or any target changing event) occurs. When such an event occurs the following actions take place (assume that the layer that the view is in is already front most):

- User clicks in a view
- The view's `WantToBecomeTarget()` method is called
- If the view want's to become the target, call `SetTarget(this)` on the view.
- `SetTarget()` calls the current target's `WillingToResignTarget()` method.
- If the current target is willing to resign, call the current target's `ResignTarget()` method, then the new target's `BecomeTarget()` method, and finally set the system target.

The default implementation returns `FALSE` from `WantToBecomeTarget()` and `TRUE` from `WillingToResignTarget()` so you will have to override these methods in your view subclass.

It is possible that you want an object other than the view to become the target when the user mouse down's. The best place to reset the target is in the view's `BecomeTarget()` method. You should override `BecomeTarget()` and set the target to whichever object is appropriate, e.g. a command object, or a selection object.

[We need to revisit the target policy mechanism, and move as much of it out of the event server and layer server and into the application framework as possible.]

Methods You May Want To Override

`void MResponder::WantToBecomeTarget()`

Before the target is changed, the new target is first asked if it wants to become the target. The default behavior is to return `FALSE`. Override this method if you want one of your responders to be able to become the target. You do not have to call the inherited method.

See the chapter on the target mechanism for more information.

`void MResponder::WillingToResignTarget()`

Before the target is changed, the old target is first asked if it is willing to resign the target. The default behavior is to return `TRUE`. Override this method if you want to want to do some checking before releasing the target. This could be used, for example, to perform edit checking before allowing the target to change. You do not have to call the inherited method.

`void MResponder::BecomeTarget()`

Override this so that your responder object gets informed when it becomes the target. You do not have to call the inherited method.

`void MResponder::ResignTarget()`

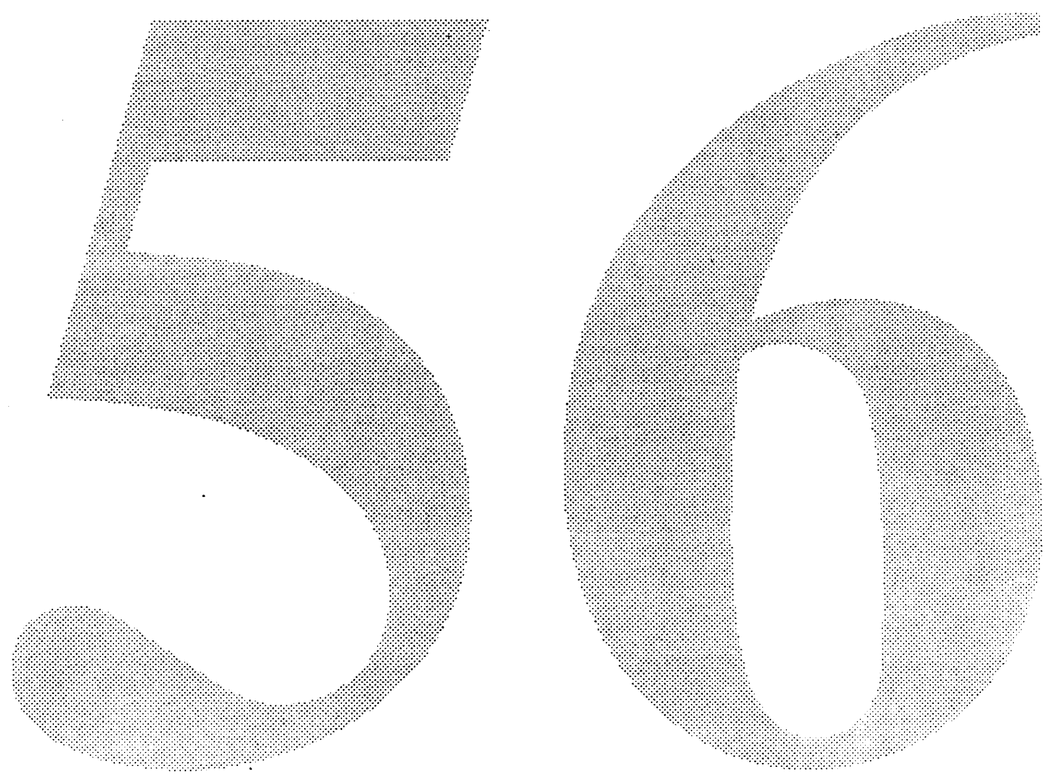
Override this so that your responder object gets informed when it is no longer the target. You do not have to call the inherited method.

`void MResponder::DeactivateTarget()`

Override this method if the responder can be a target, and you want to know when it is deactivated. Deactivation occurs when the layer in which the target is placed is no longer the front most layer. You do not have to call the inherited method.

```
void MResponder::ActivateTarget()
```

Override this method if the responder can be a target, and you want to know when it is activated. Activation occurs when the layer in which the target is placed is becomes the front most layer. You do not have to call the inherited method.



Trackers

Architecture

Trackers are objects that receive time while the user is tracking a user interface object. Tracking typically involves an **input device** such as a mouse, but may result from a **user script**.

When an event occurs, the responder that handles the event decides whether tracking is required, and if so creates a tracker object. After instantiation, the tracker object is told to `StartTracking()`, whereupon the application framework places the tracker on a **tracker queue**. Once on the queue the tracker receives time synchronously, i.e. within the application framework's main thread. All trackers on the tracker queue receive time, either periodically, at a frequency of their choosing, or based on incoming events, like "MouseMoved" events, until they receive a tracking completion event.

The application framework's main event loop blocks until either an event arrives in the event queue, or until a tracker needs time — events have priority over trackers meaning that they get distributed before tracker's get time. When an event arrives from a device that is currently being tracked, it is given directly to the tracker. The tracker looks at the event and decides if it is a **tracker completion event**. If it is a completion event, such as a mouse up, the tracker calls its own `TrackLastTime()` method, and sets it phase to `kTrackCompleted`. If it is not a completion event, the tracker calls its own `TrackContinue()` method, and returns.

It is possible to have multiple trackers tracking simultaneously. Each tracker is given time based either on the tracking frequency, or whenever an event for the device it is tracking enters the event queue. Allowing multiple, simultaneous, trackers as well as the distribution of events during tracking has the potential for creating inconsistentcies. For example, if while dragging a window, the user were to execute a command key to close the window, you might wind up dragging a non-existent window. [There is currently no framework in place to help you from shooting yourself in the foot, but I promise to think about it.]

Usage

When one of your handlers receives an event that should be tracked, you may either instantiate a tracker object and call its StartTracking() method, or if MTracker has been mixed in to an object, you may just call its StartTracking() method. After starting tracking, you should set a flag in the object being tracked so that the event handler will not allow tracking to occur with another input device until tracking has completed.

There are four tracker methods that get called by the tracking framework — `TrackFirstTime()`, `TrackFirstContinue()`, `TrackLastTime()`. `TrackFirstTime()` is called with the originating event, and you will probably want to record whatever initial state is important to you. `TrackContinue()` is the work horse routine, and will get called either periodically, or whenever a still **tracking event** occurs, such as a "MouseMoved" event. `TrackLastTime()` is called when the **tracking completion event** is received.

The calls `TrackFirstTime()` and `TrackContinue()` both return a tracker. Typically the return value is the current tracker, but it is possible to return a different tracker, allowing you the flexibility to switch trackers in midstream. This has interesting and powerful consequences. If you are tracking multiple objects, say a hierarchical menu, or are connecting blocks together with a wiring tool, each object knows how to track itself. You don't want to try to build the tracking of every possible object into one tracker, it

is much better to pass the tracker "baton" from object to object, allowing each to track itself (polymorphic tracking!). To do this requires a tracker passing protocol. Such a protocol is fairly simple, and is documented below.

The tracking framework handles several things for you. First and most importantly, after the call to `TrackFirstTime()` and between each call to `TrackContinue()`, the framework calls the initiating event's `UpdateEvent()` method. This forces the device that created the event to update the information in the event to the current state, e.g. the current location of the device's cursor. Your tracker should never query the device directly because there is no guarantee that what you want to ask is available. For example, it is easy to mimic mouse events with a script or the keyboard, and if you assumed that the device that created the "MouseUp" event was a mouse you could cause a run-time error.

Another important task that the tracking framework handles is **tracking completion**. While tracking, all events created by the device being tracked are sent directly to the tracker, by calling its `HandleEvent()` method. `HandleEvent()` decides whether the event should cause tracking to stop. If tracking should stop, `HandleEvent()` calls `TrackLastTime()` with the completion event, and sets the phase to `kTrackCompleted`. Seeing a phase of `kTrackCompleted`, the tracker framework will remove the tracker from the tracker queue and delete the tracker if necessary[5].

You will rarely subclass directly from TTracker. Instead you will subclass from TStdMouseTracker or TStdMultiClickMouseTracker.

The class **TStdMouseTracker** implements the standard mouse tracker behavior in its HandleEvent(). It looks at the incoming event and if it is a still tracking event like "MouseMoved" it calls TrackContinue() and returns. If the event is a "MouseUp" it calls TrackLastTime(), unbinds the device from the tracker so no further events get sent to the tracker, and sets the phase to kTrackCompleted.

The class **TStdMultiClickMouseTracker** implements the ability to track through mouse clicks, like the polygon tool in MacDraw. Tracking stops either when a "DoubleClick" event is received, or you can stop tracking by returning `TRUE` from the call to `CheckEvent()`. In the case of the polygon tool you would rturn `TRUE` when the user has clicked near the start of the polygon thereby closing the polygon.

Methods You May Want To Override

`TTracker* TTracker::TrackFirstTime(const TEvent &event)`

You must override this method. Store any initial state that you need, and perform any initial hiliting.

You should only query the `event` parameter for information. Because it is easy for devices to synthesize events from other devices (e.g. mouse events by a DataGlove), and because user scripts will also be able to synthesize events, you must be very careful to verify the *real* type of device before querying it. Asking a user script for its cursor location could have runtime consequences.

The return value from this method is a `TTracker*`. Typically you will return this. If you want to change the tracker, you may return any other tracker. You are responsible for deciding whether to delete the current tracker, or whether you should maintain a reference to the current tracker in the new tracker so control can be returned quickly rather than having to ask an object to reinstantiate the tracker.

If you return another tracker, it is your responsibility to determine whether the tracker's `TrackerFirstTime()` method needs to be called, and if so to call it.

5. Trackers that are mix-ins to other objects probably shouldn't be deleted, but those that are created on the fly probably should. The method `DeleteOnCompletion()` returns `FALSE` by default, if your tracker should be deleted upon completion, you should return `TRUE`.

TTracker* TTracker::TrackContinue(const TEvent &event)

You must override this method. Unlike MacApp, which has three methods — TrackConstrain(), TrackFeedback() and TrackContinue() — here there is only one, TrackContinue(). Many people had a hard time deciding how to split their tracking code into the three methods, so we make one call, and let you either do all your tracking there, or you can add methods like TrackConstrain() and TrackFeedback() if that makes sense in your case.

You should only query the event parameter for information. Because it is easy for devices to synthesize events from other devices (e.g. mouse events by a DataGlove), and because user scripts will also be able to synthesize events, you must be very careful to verify the ***real*** type of device before querying it. Asking a user script for its cursor location could have runtime consequences.

The return value from this method is a TTracker*. Typically you will return this. If you want to change the tracker, you may return any other tracker. You are responsible for deciding whether to delete the current tracker, or whether you should maintain a reference to the current tracker in the new tracker so control can be returned quickly rather than having to ask an object to reinstantiate the tracker.

If you return another tracker, it is your responsibility to determine whether the tracker's TrackerFirstTime() method needs to be called, and if so to call it.

void TTracker::TrackLastTime(const TEvent &event)

You must override this method. The event passed to this method is the tracking completion event. For example, if you are using a TStdMouseTracker you will continue to track through "MouseMoved" events, and only stop when a "MouseUp" event is received. The "MouseUp" event is the event that is passed to this method.

After TrackLastTime() is called, the phase is automatically set to kTrackCompleted, and the tracker is removed from the tracker queue. If the tracker's DeleteOnCompletion() method returns TRUE, then the tracker will be deleted.

void TTracker::HandleEvent(const TEvent &event)

If you are subclassing from TStdMouseTracker or TStdMultiClickMouseTracker, you do ***not*** want to override this method.

This method is called whenever an event from the device being tracked is distributed from the event queue. You should look at the event name, and decide if this is a tracking continue event or a tracking completion event. Your code should look something like this:

```
void TStdMouseTracker::HandleEvent(const TEvent &event)
{
    static TToken eventCompletionName("MouseUp");
    static TToken eventContinueName("MouseMoved");

    TToken eventName;
    event.GetString(eventName);
    if (eventName == eventCompletionName)
    {
        // Stop the tracking
        TrackLastTime(event);
```

```
        // Unbind the source here so that no more events get sent here.
        event.GetSource()->UnbindTracker(this);

        // This tells the tracker queue that it should remove the tracker
        // from the queue rather than giving it any more time.
        SetPhase(kTrackCompleted);
    }
    else if (eventName == eventContinueName)
    {
        // Keep tracking on MouseMoved event
        TrackContinue(event);
    }
    else
    {
        qprintf("Oops, TStdMouseTracker::HandleEvent got an event we don't
                understand, name =");
        eventName->PrintName();
        qprintf("\n");
    }
```

Methods You May Want To Call

`void TTracker::StartTracking();`

Call this when you want the tracker to start getting time. The tracker will receive time either periodically, or whenever a tracking continue event, such as "MouseMoved", occurs. The tracker can only be stopped after its phase is set to `kTrackCompleted` which should only be set by the tracking framework.

`TView* TTracker::GetView() const;`

This gets the view the tracker is currently tracking in. Since it is protected, it is only available to the tracker.

`void TTracker::SetView(TView *view);`

This sets the view the tracker is currently tracking in. This should only be set by the tracker.

`void TTracker::GetFrequency(TTime &time) const;`

This gets the current tracker frequency. This method has no meaning for trackers that are based on tracking continue events such as the TStdMouseTracker.

`void TTracker::SetFrequency(const TTime &time);`

This changes the tracking frequency after the next time the tracker gets time. Of course this method has no meaning for trackers that are based on tracking continue events such as the TStdMouseTracker.

```
TrackPhase TTracker::GetPhase() const;
```

There is really no reason to ever call this since you know the phase by which tracker method is called, TrackFirstTime(), TrackContinue() or TrackLastTime(). The tracking framework calls this to see if the tracker should be placed back on the tracking queue.

```
void TTracker::SetPhase(TrachPhase phase);
```

This should only be called inside HandleEvent(). Calling it at any other time could have interesting consequences.

Tracker Passing Protocol

There are several situations where you will want to track among a group of two or more objects. Examples include hierarchical menus, or columns in a page view, or blocks being connected with a wiring tool.

You will need to create a tracker passing protocol. This protocol is not supported by the application framework, but could be if a common protocol is observed. A description of several protocols is the best way to explain how this works.

When a tracker is changed, the previous tracker is ***not deleted*** by the tracking framework. You might want to keep a pointer to the previous tracker so that you can return it at some future time, rather than having to ask that it be reinstantiated. When changed, the new tracker's TrackFirstTime() method is not called. You should either call TrackFirstTime() yourself, inside of TakeTracker(), if that is necessary. [Not a very good solution]

Hierarchical Menus

When a mouse down occurs in the menu bar, the menu bar starts tracking until the mouse moves over a menu title, at which time a menu is dropped. As soon as the mouse moves out of the menu bar, the menu bar's `TrackContinue()` method asks the drop down menu if it wants the tracker by calling its `TakeTracker()` method. The drop down menu looks at the mouse point, and if the point is in the menu, returns a tracker. The menu bar returns the new tracker from its `TrackContinue()` method, and tracking continues in the menu. Notice that the menu bar has no idea how to track inside a drop down menu, it is the menu's responsibility to track itself. This allows new menus to be used as drop down menus without having to change the menu bar's tracking methods.

If the drop down menu has an item with a hierarchical item, it would display the hierarchical menu when the item was selected. As soon as the mouse leaves the drop down menu, it first calls the hierarchical menu's `TakeTracker()` method, and if that returns NIL, calls the menu bar's `TakeTracker()` method. It continues to call the `TakeTracker()` methods until either one of them returns a tracker (indicating that the mouse entered that menu), or the mouse goes back into the menu.

Multi-Column Page Views

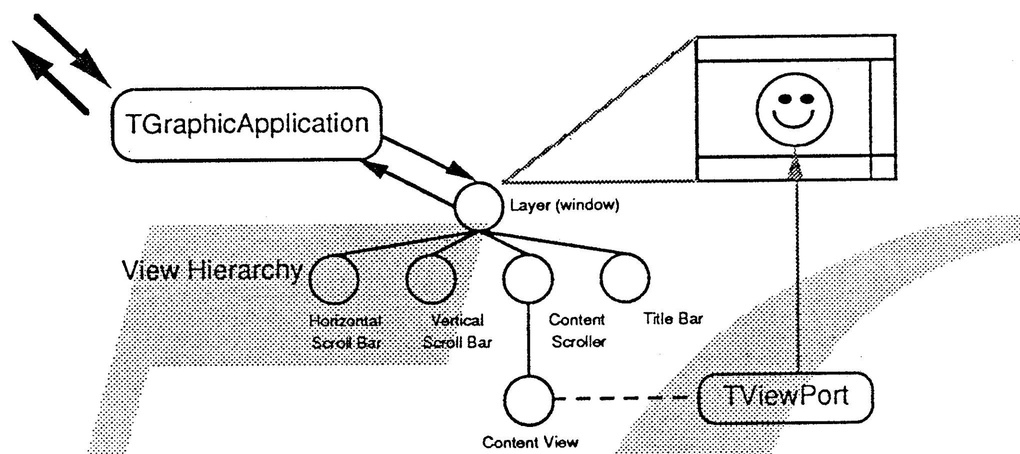
Mouse down in a view which creates a tracker, and start it tracking. If the tracker leaves the view, ask the super view to `TakeTracker()`. The super view is presumably some kind of page view, which has pointers to all columns on the page. The page view asks each column to `TakeTracker()`. If none of them respond then the mouse must be either off the page [auto scroll?] or in between the columns. As soon as the mouse enters one of the columns it returns a tracker from the `TakeTracker()` call, which becomes the new tracker.

Connecting Blocks with a Wiring Tool

Same as for a multi-column page view. Start tracking with the first block, then when the mouse leaves the block ask the super view to `TakeTracker()`. The super view will in turns call each of the other block's `TakeTracker()` method. When one of them returns a tracker you know the mouse has entered it and you can continue tracking with it

Views

Architecture



The View System decomposes an application's display into a hierarchy of display areas called views. Each view acts as a virtual piece of paper which an application can draw on regardless of where it is positioned on the screen or within the hierarchy.

Views are like classic Macintosh windows in that they act as virtual drawing surfaces. They are unlike windows in several other aspects, however. First, they form a hierarchy of arbitrary depth, unlike the single-level window system. Second, a simple view is a more primitive object than a window; it does not have a structure region, title bar, close box, or other graphic adornments. It's just an area that can be drawn in (think of it as *all* content region). Also, views are objects: their function is changed by overriding the basic TView class rather than by writing a defproc. Unlike windows, views make use of the full power of Albert: their coordinate systems can involve arbitrary transformations, not just translation, and their boundaries can be described by an arbitrary Albert area. Finally, unlike windows, views cannot be drawn in directly, because they do not have a graphic channel (GrafPort) built into them. This is a consequence of the preemptive multitasking in Opus/2, as will be explained in the Usage section below.

Each view hierarchy has a topmost view, called a layer. Layers are the units through which the desktop is shared between applications under Opus/2. They are managed by the Layer Server, which is a system wide resource. The Layer Server acts as a primitive, single level window manager, allocating screen real estate between different layers and different applications (see the "Pink Toolbox Architecture" and "Layer Server" documents). There are many possibilities for connecting windows, layers, and applications. There could be one or more than one layer per application, and one or more than one window per layer. These are policy decisions which do not affect the architecture of the View System or the Layer Server. The TGraphicApplication class acts as an interface between the Layer Server and the View System, sending and receiving the necessary Opus/2 IPC messages to keep the two coordinated.

Because of the preemptive nature of Opus/2, there are multiple resources which need to be synchronized for concurrent access by competing tasks. These are the View System itself, the layer, and the graphic device being drawn on. Although there is at most one view system per Opus/2 team, it still needs to be shared among multiple tasks (threads) on that team. In order to protect the developer from having to do a lot of explicit concurrency control, the View System may normally only be used from the main thread of the application framework (a.k.a. the interface task), the same thread that distributes events, and which gives time to trackers and idlers. Any task may draw in a view, but only the interface task can modify view attributes, such as size or position, or create and delete views.

Any task other than the interface task which wants to accomplish these ends can do so by posting an event to the application's event queue. The responder which receives the event (which could be a view, since views are responders) can make any call it wants, since responders execute in the interface task.

The layer itself is synchronized separately. The visible region part of the layer information must always be up to date to prevent an application from drawing on an incorrect portion of the screen. The interface with the Layer Server is carefully controlled inside the View System to prevent an errant application from keeping the layer semaphore and locking up the whole system.

Finally, the Albert graphics system synchronizes access to the frame buffer on a device by device basis; the Toolbox is not involved.

Usage

Types of Views

The primitive view system types are TView and TViewPort. There are also many subsidiary types:

- TLayer, mentioned above, is a subclass of TView which is at the root of the view hierarchy and which corresponds to a Layer Server layer.
- TScroller is a special kind of view which can translate its coordinate system. This allows the effect of scrolling to be achieved.
- TSplitter is a view which can show its subviews multiple times, to show a split pane.
- TTransformer is like TScroller, but allows more general transformations than translations, such as zooming and rotation.
- MPrintable is a mixin class which allows a view's contents to be paginated so that it can be printed.

Drawing into Views

There are two options for drawing in a view. The first is to draw synchronously when an update event is received; this is done by overriding the `DrawSelf(TViewPort *viewPort)` method of TView. If you decide to respond to the update in the `DrawSelf()` routine, you must use the `viewPort` supplied. This approach is suitable when there is no need for animation or to draw asynchronously. It doesn't work well when drawing is a lengthy process, as that would tie up the interface task, freezing the user interface while drawing takes place.

The other option is to allocate a TViewPort into which you may draw asynchronously. A view port is a descendant of the TGrafPort class defined by Albert. Every view port is associated with exactly one view and exactly one task. A view can have many viewports associated with it, however, and a task can also have multiple viewports at once. Once the view port is allocated, the owning task can draw in the view whenever it wants, although the developer must synchronize simultaneous drawing to the same view. [???]

The view port can be used either for animation or to process updates in background. The latter is accomplished by overriding `TView::RefreshInterior()` to return `FALSE`, which indicates that the update was not performed synchronously. When the application is ready to start processing the update (e.g. after creating the task to perform the update), it calls the view port's `BeginUpdate()` method; when drawing is completed, `EndUpdate()` must be called.

Building View Hierarchies

When designing the contents of a window it is useful to break the window down into logical subdivisions. For instance in a page layout program, or word processor, there are columns and pictures along with user interface controls such as scroll bars. In a dialog box there are areas with editable text, areas of scrolling items, areas of static pictures and text, and user interface items such as check boxes and buttons. Using the TView class it is possible to match your logical subdivision with a physical subdivision.

The benefits of matching the physical to the logical is that the handler's for each of the separate items in a window can be self-contained. Check boxes can handle their own checking and unchecking. Buttons know how to click themselves, text boxes know how to edit the text entered in them, and a dialog box can impose an order on top of all of these items, for instance tabbing properly between them.

There will be cases where you can use the TView class directly, but more likely you will want to subclass and implement behavior specific to your type of view. In the following sections you will encounter window, menu, editable and static text boxes, button and other user interface classes that descend from TView. You will need to decide from which of these classes to override.

Unresolved Issues

There are still some unresolved issues in the architecture of the View System. These include:

- Support for rubber-banding or sprites. There are some kinds of user interface activities which involve drawing on *top* of views which may be in the process of actively drawing. A good example is dragging an icon between windows, as in the Finder. Some of the windows the icon passes over may have tasks actively drawing in them. This needs to be coordinated to prevent the screen from being corrupted. We are examining several alternative implementations.

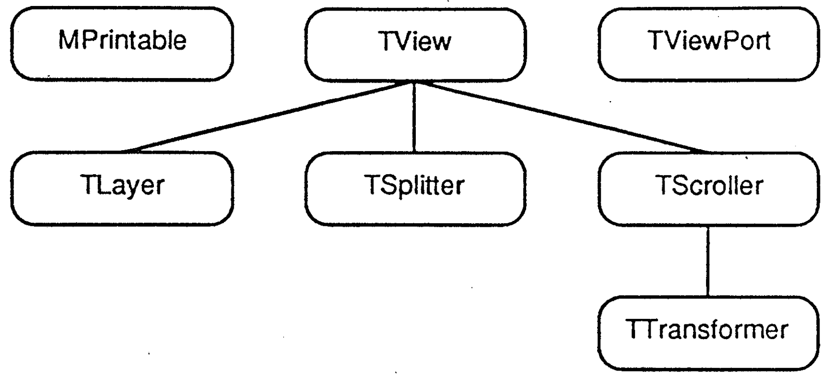
- There is no support for color palettes, pending the definition of this kind of function in Albert.

- Support for animation, off-screen buffering, saving pixels (as for menus), and device management all await further clarification of the Albert spec. Some details given below may change as the Albert spec changes.

- The performance of the synchronization mechanisms we have chosen will have to be carefully measured to ensure that the overhead is acceptable.

- Support for networked views.

Class Diagram



TView

Methods You May Want To Override

```
void          RefreshBackground(TGrafPort *);
Boolean       RefreshSelf(TSeed, TGrafPort *);
void          DrawSelf(TGrafPort *);
void          Activate();
void          Deactivate();
Boolean       AcceptsActivatingEvent() const;
void          DrawSelf(TGrafPort *);
void          RefreshSelf(TGrafPort *);
```

Methods You May Want To Call

```
Boolean       IsVisible() const;
void          SetVisible();
MResponder*   GetTarget();
void          SetTarget();
void          ContainerToSelf(TGPoint &) const;
void          SelfToContainer(TGPoint &) const
void          GlobalToSelf(TGPoint &);
```

```
void        SelfToGlobal(TGPoint &);

Boolean     Contains(const TGPoint &) const;

Boolean     ContainsInSelf(const TGPoint &) const;

void        GetBoundary(TArea &);

TGPoint     GetLocation() const;

void        SetLocation(const TGPoint &);

TGPoint     GetSize() const;

void        SetSize(const TGPoint &);

void        AddViewAtBack(TView *);

void        AddViewAtFront(TView *);

void        AddViewBefore(TView *view, TView *before);

void        BringForward(TView *);

void        BringToFront(TView *);

void        MoveBefore(TView *view, TView *before);

void        RemoveView(TView *);

void        SendToBack(TView *);

void        SendBack(TView *);

TView*      FrontView();

TView*      BackView();

Boolean     GetDrawClipped();

void        SetDrawClipped(Boolean);

//void      InvalidateClipped(const TArea &);

Boolean     IsSubView(const TView *);

TView*            FindSubView(const TGPoint &);

void        SubViewMoved(TView *view, const TGPoint &oldLoc, const TGPoint
&newLoc);

void        SubViewChangedBoundary(TView *);
```

```
void        SubViewChangedVisibility(TView *);
```

Windows

Architecture

Windows are views that appear on the Pink desktop. Typically each window appears in a separate layer, but it is possible to place several windows in a single layer.

Windows know how to drag, grow, and size themselves, responding to a variety of events such as "Move", "Drag", "Zoom", and "SetAppearance". Every window has a title, but need not have a title bar, allowing windows to be found by name.

Each window has a single **subview called the content frame** — this content frame is owned by the window. Inside the content frame appears the **content view** — a view owned by your application. The content frame is used by the window to ensure that your content view does not draw onto the window border or frame. The content frame is created by the window, the content view is created and inserted into the content frame by you.

It is inside the content view that you will place the views used by your application. If you have a simple application that requires but a single view, you can display it in the content view directly. If you have a more complex application, a page layout program for example, your content view will be a page layout view — one of your own making, or one supplied by the ZZText portion of the Toolbox — inside of which you will place galley or column views.

Class Diagram

See first page of this chapter.

Usage

As with all views and responders, you should only call window methods from the main application thread, i.e. at event distribution time, while tracking, or while performing idler tasks.

Parameters to the window's constructor allow you to:

- Place the window in a separate layer or not
- Use the standard title bar or not
- Place the window into one of six layers
- Place the window either front or back most amongst other windows in the layer

If you do not place a window in a separate layer, you must place it inside some visible view to make it visible. You may either place it directly in an existing layer, or you may place it in any other visible view, e.g. another window.

If you choose not to use the standard title bar, you may either display the window without a title bar, or provide your own title bar.

By default, new window's are placed in the **document layer**. Supplied window subclasses, e.g. menus, menu bars and windoids will place themselves in their correct layers, so you will typically have no need to override the `LayerKind` parameter. The kinds of layers in their order of appearance (from top to bottom) are:

- Menu Bar Layer

- Menu Layer
- Alert Layer
- Torn-off Menu Layer
- Windoid Layer
- Document Layer
- Desktop Layer

Methods You Have To Call

```
void TWindow::Init()
```

Because of C++ limitations, it is impossible to correctly set up several window parameters correctly until *after* the constructor has been called. After creating a window you must call the `Init()` method to allow the window to properly initialize itself. A window will not become visible until after its `Init()` method has been called.

Methods You May Want To Override

```
TContentFrame* TWindow::CreateContentFrame();
```

Override this to create your own content frame, which must subclass from TContentFrame. The standard content frame uses the parameters `borderWidth` and `borderHeight` to create a rectangular content frame. The content frame size is automatically changed whenever the window changes size.

Remember that the content frame is owned by the window, and must never be modified arbitrarily.

```
void TWindow::InvalidateBorderForResize(const TGPoint &oldSize, const TGPoint &newSize);
```

If you are creating your own special window you will want to override this method to perform any special invalidation as a result of window resizing. Whenever the window changes size, the content frame's size is also changed automatically. In the standard window there is a one pixel border inside the content frame, surrounding the content view. That border may not be included in the update region, so it needs to be included in order for it to be erased properly in response to an update event.

Methods You May Want To Call

```
TContentFrame* TWindow::GetContentFrame() const;
```

This call allows you to get the current content frame. Along with `SetContentFrame()` it is possible to change content frames on the fly (though why you would want to do this is not immediately obvious).

```
void TWindow::SetContentFrame(TContentFrame *contentFrame);
```

If the new `contentFrame` is `NIL` do nothing. If there is a current content frame, all subviews, if any, are removed, and added to the new content frame, the current content frame is deleted, and the new `contentFrame` is installed.

```
TView* TWindow::GetContentView() const;
```

Get the current content view, if any. The content view is owned by your application.

```
void TWindow::SetContentView(TView *contentView);
```

Removes the current content view, if any, from the content frame, and installs the new `contentView`. Does not delete the current content view.

```
void TWindow::GetTitle(TText &title) const;
```

Return the current title in `title`.

```
void TWindow::SetTitle(const TText &title);
```

Set the title to `title`.

```
void TWindow::SetSizeUsingContentFrame(const TGPoint &contentFrameSize);
```

Set the size of the window by specifying the size of the content frame to be `contentFrameSize`.

Menus

Architecture

Menus are just windows that reside in a menu layer. The menu layer is below the menu bar layer, but above the alert layer.

Menus have pre-defined trackers that make it easy for you to specify Pink style menus, both textual and graphical.

Controls

Architecture

To be filled in later.

56

Cher

Document Architecture

56

CHER ERS: The Next Generation

Arnold Schaeffer
x8117

Larry Rosenstein
x8123

Cher is the Document Architecture for Pink. Its main goal is to raise the base level of Pink applications by enabling several new features such as multi-level Undo, hypermedia linking, annotations, real-time collaboration, and content-based retrieval.

Although many of these features are available in existing Macintosh applications, the lack of system-level support reduces their benefit to users, and limits their implementation.

Cher provides services in the areas of: (1) collaboration, (2) hypermedia linking, (3) eternal undo, and (4) retrieval, content-based. To the extent possible, Cher does not specify any policy or user interface decisions. For example, Cher does not specify how links appear to the user, or how retrieval queries are formulated. We expect that the low level mechanisms provided by Cher will be used to implement a variety of high-level features.

56

Introduction

Cher is the Document Architecture for Pink. Its main goal is to raise the base level of Pink applications by enabling several new features such as multi-level Undo, hypermedia linking, annotations, real-time collaboration, and content-based retrieval.

Most of these features are available in some existing Macintosh application. Unfortunately, these are isolated cases, because there is no system-level support for them.[1] This lessens their impact because it reduces the synergy between applications. For example, only a few applications support multi-level Undo, so users can't count on having this feature available.

In addition, the lack of system support for these features limits their implementation. For example, there are applications that allow users to annotate static representations (pictures) of any document, but not the "live" document itself. The content-based retrieval applications have trouble accessing the contents of document because each application has a custom file format. Also, once the application finds a document, it is difficult to do anything with it. There's no system-level support for opening the document, for example.[2]

In the Pink system, the architecture for implementing these features is defined by Cher. The result is that these features will be standard in every Pink application. Cher provides services in the areas of: (1) collaboration, (2) hypermedia linking, (3) eternal undo, and (4) retrieval, content-based.

To the extent possible, Cher does not specify any policy or user interface decisions. For example, Cher does not specify how links appear to the user, or how retrieval queries are formulated. We expect that the low level mechanisms provided by Cher will be used to implement a variety of features.

What Does Cher Provide?

As mentioned above, Cher provides services in four areas: (1) collaboration, (2) hypermedia linking, (3) eternal undo, and (4) retrieval, content-based. These areas are described below. This is primarily a user-oriented description of Cher. The programmer's view is described in the following section.

C is for Collaboration. There are many kinds of computer-supported collaboration. Screen sharing is popular on the Macintosh, because it is relatively easy to implement and can be put to many uses. Its main disadvantages are that some applications draw directly to the screen (complicating the implementation) and the large bandwidth required to transmit all drawing operations from one machine to another.[3]

Screen sharing is an example of simultaneous, real-time collaboration. Cher provides support for a different kind of real-time collaboration. This operates at the level of changes to the document, rather than changes to the screen. Potentially, this will be more efficient because the amount of data needed to specify a document change is usually less than the amount needed to update the screen.

Cher does not specify any synchronization between collaborators. One possibility is no synchronization

1. System 7 does add some support for these features.
2. Again, System 7 will fix some of these problems. In particular, a retrieval program should be able to send an AppleEvent to the Finder and open the found document(s).
3. Since this form of collaboration is implemented in the graphics system, it is beyond the scope of Cher. Pink will probably support screen/window sharing in its graphics system.

("free-for-all"), in which the participants would be able to make changes at any time. The participants would be forced to adopt some ad hoc conventions to prevent chaos, but these wouldn't be imposed by the system.

It is likely, however, that the user interface above Cher will implement some kind of protocol for passing control from one user to another. For example, there would be one person who "has the floor" and can pass control to another collaborator.

Alternatively, each user could attempt to acquire a network semaphore when she started to pull down a menu or modify the document. This provides synchronization without any explicit user action.

It is also useful to have asynchronous (i.e., non real-time) collaboration. One form of this is the ability to annotate a document. Cher provides low-level support for annotations through its linking mechanism (described below).

H is for Hypermedia Linking. In Cher, a link is a bidirectional connection between anchors.[4] The meaning of an anchor is application-specific, but in most cases an anchor identifies a sticky selection. An anchor is sticky in that it always refers to the same data regardless of the user's editing changes. For example, if the anchor refers to a word within a text block, it always refers to that word, even if the text around it changes.[5]

Once the user creates a link, she can operate upon it. First, the user can navigate from one end of the link to the other. In general, this involves opening the document containing the target anchor, scrolling the anchor into view, and highlighting the corresponding selection. Applications can change this behavior; for example, navigating to a sound document may simply play the sound without opening the document.

It is also possible to transfer data across the link in either direction. The semantics of transferring data is (roughly) equivalent to copying the source data, navigating to the destination anchor, and replacing the existing data with the transferred data. It makes no difference to Cher whether the data is pushed or pulled through the link (i.e., whether the source or destination initiates the transfer).

Cher also allows one application to send an arbitrary message across a link. This will allow cooperating applications to implement custom features using the same basic linking mechanism.

The straightforward use for Cher's low-level linking mechanism is to allow users to create links between documents, navigate those links, transfer data across them, etc.

This isn't the only use for links, however. We expect that links will be used to implement other application features. In these features, the fact that links are created and manipulated will be transparent to the user.

For example, Intermedia uses the low-level linking mechanism to implement an annotation facility. The user selects a part of the document and chooses Create Annotation. Intermedia creates a link between the selection and an annotation document, and transfers the selected part of the document to the annotation. The user can modify the annotation to suggest a change to the document.

If the author decides to incorporate the suggested changes, she can choose the Incorporate Annotation command. This simply transfers the proposed change across the link into the document. Neither the author nor the annotator is aware that linking is involved.

---

4. The design of linking is heavily influenced by the Intermedia project.[Meyrowitz 1986]
5. If the user changes something within the anchor, then the expected thing happens; the text associated with the anchor changes.

Another transparent use of linking could be to simulate the functionality of the System 7 Publication Manager. Suppose a user wants to incorporate a drawing to a word processor document. She can select the drawing and choose the "Publish" command, select a position in the word processor document, and choose "Subscribe."[6] This would create a link between the drawing and the word processor document. The system would attach an attribute to the link to indicate the publish/subscribe nature of the link. Note that the existence of the link is invisible to the user.

Once the link is in place, the system can provide a command that opens the publishing document or any subscriber. The system can automatically push the data across the link when the published document changes.[7]

This kind of annotation implemented in Intermedia uses separate documents and windows for the annotations. An alternative is to use a kind of marginal note. This is already planned for Hoops, and is being explored by a group at Carnegie Mellon University[Morris et. al. 1988]. The advantage of this approach is that each annotation does not require a separate window. If the system support multiple columns for notes, then it is easy to compare the annotations made by different users. Also, if Hoops intends to support marginal notes, then perhaps this should be a generic part of Pink to promote consistency.[8]

is for Eternal Undo. In most Macintosh applications, the undoable command is precious, since only the last change can be undone. In Cher we use the same kind of command objects as MacApp, but expect to save as many command objects as possible.

This decision has several benefits. First, it isn't as important to be choosy about what commands are undoable. For example, in existing Macintosh applications, changes to the selection are not undoable, even though selections can be difficult to create. In a drawing program, the user can spend much time selecting the right combination of shapes and lose everything with an extra click.

If the system supports only one level of undo, then it is unwise to save a selection change if it means forgetting about the last Cut command, for example. With multiple levels of undo, it is feasible to save selection changes.

With this change comes the added burden of providing a good way to visualize and navigate the list of commands.[9] This is especially true if selection changes are included, because it will be easy for the user to create hundreds of commands.

Another benefit of multi-level undo is increased reliability. If every command is saved, then it is possible to replay those commands in the event of an application or system crash. Cher will use the concurrency control and recovery classes (Credence) to save command objects in a robust manner.

Note that Cher maintains a linear list of command objects. This means that undo returns the document to some previous state. In theory, it is possible to enhance the undo model to allow the user to selectively undo commands (i.e., undo a Cut command but keep all subsequent commands intact).

The problem is that commands are not independent of one another. A command that copies a shape is

---

6. I'm just guessing at the current interface in System 7.
7. To integrate a Pink application with a System 7 application (running under the Blue Adapter) we would write a Pink program that manages the System 7 publication files, and allow the Pink application to link to the publication file.
8. Thanks to Rob Chandhok for pointing this out.
9. This isn't the only place where a good navigation mechanism is required. We will also need a way to visualize the set of explicit links between documents, for example.

dependent on the command that first created the shape. These dependencies would complicate the user interface to undo, as well as the underlying implementation.

The best solution is to integrate the undo and scripting mechanisms, for example to automatically create a script of everything that is done. The user can then edit the script to remove arbitrary command, rearrange command, etc. and execute the script. This gives users the maximum flexibility and control.[10]

R is for Retrieval, Content-based. Increasingly, users have more and more information available on their computers. Local hard disks are getting larger, and there are many CD-ROMs available that contain hundreds of megabytes of data.

It is impossible to browse through this data without some assistance from the system. In the Macintosh, the only standard tool is Find File, which located document based on their name. System 7 will provide a faster version of this integrated within the Finder. Third party developers now provide tools that go beyond Find File and search for documents based on their content. The linking mechanism provided by Cher can be used to navigate from one document to another, but requires that someone has previously set up the links.

It is important that these retrieval tools be integrated into the system. There's little point in locating documents if the user can't do anything with them. The third party content-based retrieval tools on the Macintosh get no system support in examining the contents of the document, or even opening the document with the appropriate application. Similarly, On Technology had to go to great lengths to automatically index documents when they change.[11]

Cher will provide a generic retrieval framework. The framework will handle both indexing and queries. Although Pink will ship with a default indexing and query package, we expect that users will want the ability to plug in new search packages into the basic framework. The point of designing a framework is so that the background indexing mechanism and query user interface are the same regardless of the underlying retrieval technology.

The framework will provide for automatic, background indexing of documents when they are added to a volume or changed. A retrieval system is worthless if only some of the documents are indexed, and it is unreasonable to place this burden on the user.

We would like to develop a generic query front-end so that if a user does install a new retrieval engine she doesn't have to learn a new front-end.

Concepts

Writing an application that works with Cher requires a slightly different perspective than writing an existing Macintosh application. For example, implementing sticky selections generally requires a different implementation for low-level data structures. This section describes some of the concepts of Cher, in order to give you an understanding of what is involved and how you approach a Pink application vs. a

10. Of course, the specification of selections in the scripting language must be sufficiently robust to allow this kind of editing. Otherwise, the user will have to do more work than just rearranging items in the script.
11. At least they made the effort to do this. The NeXT requires that the user explicitly index documents in its Digital Librarian.

MacApp application.

Models, Views, & Presentations

Unlike MacApp, Cher defines an architecture for managing a document's data. The architecture doesn't define actual data formats (either in memory or on disk), but rather defines the protocol between various objects in the application.

The principal object is the Model. A model contains the actual data of the document. Most Pink developers will create an application-specific model class, although it is likely that Pink will include some standard model subclasses.

The image of a model is displayed within one or more views. The Pink View System is described separately, but the essential details are that views define their own local coordinate system, and windows are composed of a hierarchy of views.

Although there is a relationship between models and view, it is advantageous to make the coupling be as loose as possible. This results in more flexibility. For example, it should be easy to display the same model in different views. It should be easy to create new kinds of views of a model. It would be desirable if a particular view class could be used with different kinds of models.

In order to accomplish these goals, we need a protocol between models and views. Essentially, when a model changes all the views that depend on that model need to be notified of the change. The views can then access the specific details of the Model and make the necessary changes on the screen.

The link between the model and view is the *presentation*. The main function of a presentation is to define the change protocol between a model and its dependent views. There are several ways this change protocol can work.

The simplest protocol[12] is one in which each change to the model is immediately propagated to the dependent views. In some cases, it might be desirable for the presentation to accumulate a series of model changes, and allow the view to request the changes when appropriate.

A second function of the presentation is to store view-related data that must be shared between views. For example, suppose we have two views of the same text model. To display the text on the screen, the view needs to compute the appropriate line breaks. If there are 2 views that require the same set of line breaks (e.g., the text is displayed in a split window), then the natural place for them is in the presentation object.[13]

Commands & Document Saving

Like MacApp, Cher uses the concept of a command object in its framework for Undo. (As described in the next section, command objects are also central to the collaboration features of Cher.)

Unlike MacApp, Cher saves more than just the last command object created. This provides the user with the ability to Undo (or Redo) many changes to the document. Given enough disk space, it will be possi-

12. This is also the only one currently implemented. We probably will implement alternatives in the future.
13. They can't be stored in the model, because line breaks are relevant only to the view. Also, there might be other views of the text with different margins that require different line breaks.

ble to undo back to a blank document.

In addition to providing multi-level undo, saving "all" command objects enhances application reliability. Cher will periodically (in the background) save command objects to disk, using the Pink concurrency control and recovery (CCR) facility. The result is that if the application or system crashes, the user shouldn't lose more than the last few commands. The command objects will be part of the user's document, so the system will automatically "replay" those commands when the document is opened.[14]

Today, users save document more often then they want, because they are afraid of crashes that would lose the entire document. Under Pink, these kinds of saves won't be necessary. The only time when the user needs to save is when she has a complete, new version of the document.[15]

For the most part, Cher's multi-level undo doesn't make implementing command object any more difficult. One difference is that applications won't be able to "get away" with non-undoable commands. Many of the current Macintosh applications when it comes to undo. Most will not undo a command such as Find and Change All, because doing so requires saving a duplicate of the document. Others don't undo command that are "easy" for the user to undo herself (e.g., font change commands in PageMaker).

It isn't possible to get away with these shortcuts in Pink. For example, if Find and Change All isn't undoable, then it is possible that a previous command no longer makes sense. (You can't undo a Paste command if the pasted text was changed by Find and Change All.) It is unacceptable to throw away all the saved commands, as applications now do.

Another difference concerns filter commands. A filter command is one that doesn't immediately change the internal data structures, but simply shows the change on the screen. The data structures are changed only when the command is no longer undoable; i.e., when the user performs the next command.

Filter command are useful when the operation can't be undone easily. For example, suppose the user selects an entire graphics document and changes the fill pattern to black. To reverse this command requires that we save the old fill patterns for each shape. Using a filter, the program simply draws the affected shapes through a "black filter." Undoing the command simply involves removing the filter.

In an environment that supports multi-level undo, the filtering technique isn't as useful. Since each command can potentially require a filter, the system would have to support multiple levels of filtering. Accessing the underlying data would be slow because of the multiple filters. For this reason, we expect that command objects in Pink applications will just save all the data needed to undo the command, even if it means making a copy of the current selection. In some case, it might be possible to compress this data to save space.

Cher also provides a framework for "long running commands." These are commands that are likely to take a long time and should be run in the background. We don't expect that the user will be able to make additional changes while a command is running, although individual applications can support this if they choose to. For the most part, the user will be limited to operations that don't change the document, including scrolling, window resizing, and switching to other applications.

There are still synchronization issues, however, because refreshing the window may require access to the model. Cher maintains a semaphore for each model to control access to it. By default, all command ob-

---

14. Applications can take advantage of this facility to speed up saving. It is possible to specify a change limit for a document. If the user makes fewer changes than the limit, then Cher will save only the new command objects. If the user makes only one change to a document, the document could be saved simply by writing out the new command object.

15. Although, it may be difficult to change peoples' habits about saving.

jects run in a separate task and Cher acquires exclusive access to the model semaphore before running the command. Interface-related tasks that require access to the model acquire the semaphore shared.

This normally forces the interface to block while a command is running; although Pink will allow users to switch applications under these circumstances. A command objects can, however, copy data out of the model and release the semaphore, which will unblock the user interface. When the command needs to store data back into the model it will reacquire the semaphore and make the change. The command object can make multiple small changes to the model so that the user can incrementally see the affects of the command.[16]

Real-time Collaboration

As mentioned earlier, Cher provides support for real-time collaboration. From the collaborators' points of view, they are working on the same document simultaneously, as if they were sitting in front of the same machine. All the users collaborating on a document share the same selection and list of commands. One user can undo commands made by another, for example.

This is implemented by forwarding command objects to a *model server*. The model server applies the command object to the document's model as well as forwarding the command to the other collaborators. It is a task running on the machine of the user who started the collaboration.

Normally, each additional collaborator (other than the one who started the collaboration) maintains a cached version of the model. When a command object is received from the model server, the application applies it to the cached model. Alternatively, the application can send messages to the model server in order to retrieve parts of the model.

When the model server first applies the command object, the command has the option of returning a different object as the result. The new object is the one that the model server forwards to the other collaborators. This allows a Paste command to be implemented by a general command that replaces one selection with another.

Because command objects are sent from one user to another, it is important that the commands be "model-independent." In MacApp, command objects would normally contain references to parts of the model changed by the command (for example). This implementation strategy isn't possible in Cher. In most cases, this requirement affects only the design of the selection objects (see below).

Selections & Anchors

The Cher architecture includes the concept of a selection object, which is not part of the MacApp framework. In MacApp, the concept of a selection is folded into the view class.

Cher uses selection objects to manipulate data in the document. The result is that most of the standard

16. Currently, semaphores in Pink are fair with respect to readers and writers. If a reader task acquires a semaphore shared and a writer attempts to acquire the semaphore exclusively (and blocks), then subsequent readers will also block. This guarantees that the writer will get a chance to makes its change. To properly handle commands, however, we need an "unfair" form of semaphore that will give the user interface priority over the background command object.

editing commands (Copy, Paste, Push Data, etc.) can be implemented to call method of the selection. This means that developers don't have to implement these command objects, but can simply use the classes defined in Cher.

Like command objects, selection objects must be independent of the specific model object to which they refer. That's because selection objects are distributed to each collaborator. Therefore, developers must implement selection objects as specifications of the selection, rather than with pointers into the model.

In the case of a text model, one possible selection specification is a pair of character offsets. In a structured graphics model, each shape must be assigned a unique id, and the selection specification is a set of unique ids.

Anchor objects are very similar to selection objects in that they refer to part of the model. The difference is that anchors must be resilient to editing changes, since by definition, an anchor is a *sticky* selection. An anchor object contains a selection object, so it is usually advantageous to make all selection objects sticky, and use an instance of the appropriate selection class in the anchor.

The implementation of graphics selections described above is sticky. The implementation of text selections, however, is not. If the user inserts or deletes text before the selection, then the character offsets must be adjusted.

There are a couple of approaches for implementing text anchors. First, the model could maintain a collection of markers that point within the text. The anchor would be a unique id that refers to a marker. When the text was changed, the appropriate markers would be updated, but the anchors would remain the same.

Another approach is to maintain an editing history for the text.[17] The anchor would contain a pair of character positions, as well as a time stamp. Each time the text was edited, the history would be updated to record the change (e.g., 5 characters deleted from position X at time T). When the anchor is used, the system would have to correct its character positions based on editing changes that happened after it was created. At convenient times, the history can be condensed and the anchors permanently updated.

Cher does not specify how anchors are represented to the user. Anchors must be selectable objects, since the user selects an anchor in order to operate on a link. Also, it is possible to copy and paste anchors. Some applications will allow users to move anchors around. We will probably define some consistent "look" for anchors so that users can recognize them in any application.

Linking

To create a link between two anchors, the user must specify the anchors. The situation is similar to copying and pasting data (the user needs to specify a source and destination), so one way to do this is to maintain a kind of "linkboard," analogous to the clipboard. The Start Link command would create an anchor out of the current selection and place the anchor on the linkboard. The Complete Link command would also create an anchor, and then create a link between the new anchor and the one on the linkboard. It is possible to choose Complete Link several times, in order to create several links that share a common anchor.

In Pink we are investigating extensions to the current Macintosh clipboard model. Two of these extensions are to support more than one item on the clipboard (as in the Scrapbook) and to support more direct

17. This is the approach used by Intermedia.

manipulation (dragging items to the clipboard as an alternative to Copy). We expect that the linking user interface will closely match the copy and paste interface.

It is also possible for an application to create links programmatically. We expect that most of the "interesting" uses of linking will fall into this category. For example, annotations can be attached to the affected parts of the document with links. Scripts can also refer to parts of a document with links.

There is only one kind of link in Cher. It is bidirectional, and supports both navigation (finding the other end of the link) and data transfer. Links also have properties, which applications can use classify links and to restrict how links are used.

For example, there could be properties that specify what the user can do with a link. It might be desirable to allow certain users only to pull data from a spreadsheet and not navigate to the spreadsheet or push data into it.

Links that are used to implement annotations will be identified by a certain property. This property will indicate that the link is part of an annotation, and that the appropriate annotation commands are enabled.

A final example is a link that indicates a master-copy relationship. This would be used to import a graph, for example, into a word processor document. Instead of copying and pasting a static representation of the graph, the user can create a link between the original graph and the copy placed in the word processor document.

Once this link is made and its role defined, then the system can provide an Edit Original command. The user could select the graph in the word processor document and choose Edit Original. This would navigate to the document containing the graph and allow the user to change it. When the changes are saved, the resulting graph would be pushed back to the word processor document. This provides most of the benefits of a component document architecture, without changing the basic document model.

References

Meyrowitz 1986 — Norm Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypertext/Hypermedia System and Applications Framework," Portland OR, Proceedings of the OOPSLA conference, September, 1986.

Morris et. al. 1988 — Jim Morris, Davis Kaufer, Chris Neuwirth, Ravinder Chandhok, "The 'Work in Preparation' (PREP) Editor: Support for Co-Authoring and Commenting." A Proposal to the National Science Foundation. November 13, 1988.

Classes

This section of the document contains class and member function descriptions. In the interest of keeping this document to a manageable size, certain conventions are followed. All classes have virtual destructors. If the destructor does anything more than release storage, it is discussed; otherwise, nothing is said. Many of the classes have getters and setters which simply do field accesses. These methods are listed in the class declaration but are not discussed in detail.

Unfortunately, class descriptions do not fully describe the connections between all of the objects. Example programs, such as TurboPinkDrawII® augment the documentation greatly.

Entitys

A `TEntityID` is a class which provides a name for a finder entity that is capable of responding to at least a minimal set of event messages which we will define (including "Launch"). This class will encapsulate information provided by the file system for naming things. For now, entities could be named by a `TText` object or by a unique id which says which entity this is on the volume. This class is a temporary class until the finder provides this functionality.

```
class TEntityID : public MCollectible {
  public:
     TEntityID(long uniqueID, const TText& name);

     virtual const TText& GetEntityName() const;
     virtual long         GetUniqueID() const;

  protected:
     virtual void         SetEntityName(TText&);
     virtual void         SetUniqueID(long);
};
```

Type Descriptions

`TTypeDescriptions` encapsulate information about a type used for type negotiation during cut/copy/paste or push/pull. Basically, the `TTypeDescription` object contains information about a specific type being published and whether that type can be abstract, concrete (that is, more than just protocol - contains data), or both.

For example, suppose we wish to cut a piece of animation from one document to another document. Part of the cut and paste process involves negotiating what types each document can understand. A document might know how to deal with an object as a baseclass but not know about the derived class. For example, suppose there is an abstract baseclass in the system for animation, `TAnimation`. There might also be a class for pictures, `TPicture`, that everybody knows about. The document containing the data might publish a list of type descriptions that looked like:

```
TTypeDescription(TToken("TMyAnimation"), kAbstractOrConcrete)
TTypeDescription(TToken("TAnimation"), kAbstractOnly)
TTypeDescription(TToken("TPicture"), kConcreteOnly)
```

The use of a type when only the abstract baseclass is known usually involves loading code from a shared library (see page 13).

```
enum TypeKind { kAbstractOnly, kAbstractOrConcrete, kConcreteOnly };
```

```
class TTypeDescription : public MCollectible {
 public:
    TTypeDescription(const TToken&, TypeKind theKind = kConcreteOnly);

    virtual  const TToken&  GetTypeName() const;
    virtual  TypeKind       GetTypeKind() const;

    virtual  void           SetTypeName(const TToken&);
    virtual  void           SetTypeKind(TypeKind theKind = kConcreteOnly);
};
```

Selections

Before performing an operation on an object or group of objects, the user must select it to distinguish it from other objects. This is known as a *selection*. There are classes in CHER to represent a selection. The mixin class, `MSelectable`, provides most of the protocol that document selections and anchors are expected to implement. The class, `TDocumentSelection`, should be used as a base class when applications are creating classes which represent the different kinds of selections supported in the application. It is important to think of selection objects as being *specifications* of what the actual selection is as opposed to the selection itself. For example, in a text document, the `TTextDocumentSelection` would refer to a range [140, 155] of characters instead of containing the actual characters in the selection object.

`MSelectable` objects contain the protocol for exchanging data between selections using cut, copy, and paste or using push/pull (on anchors). This includes the protocol for type negotiation (what types can I publish this data in, what types can I accept data in) and the protocol for actual accepting data or publishing data in the specified type.

DISCLAIMER: The selection protocol is in the process of being overhauled to accomodate cut/copy/paste of anchors and links.

MSelectable

The mixin class, `MSelectable`, provides most of the protocol that document selections and anchors are expected to implement. This protocol is used internally by CHER to implement all of the data exchange commands (cut, copy, paste, push, pull).

```
class MSelectable : public MPersistent {
 public:
    virtual  MCollectible*        GetCopyOfData(
                                     const TTypeDescription* t = NIL) const;
    virtual  void                 GetTypes(TDeque& theTypes) const;
    virtual  void                 ChooseType(
                                     const TDeque& theChoices,
                                     TDeque& theChosenTypes) const;
    virtual  TDocumentSelection*  AcceptData(const TDeque& theData);
    virtual  TDocumentSelection*  AcceptData(MCollectible* theData);
    virtual  MCollectible*        GetSelectionDataForUndo() const;
    virtual  MCollectible*        GetSelectionDataForExchange(
                                     const TTypeDescription* t = NIL) const;

    virtual  void                 SetModel(TModel* theModel);
    virtual  TModel*              GetModel() const;

 protected:
```

```
    MSelectable(TModel* theModel = NIL);
};
```

```
MCollectible* MSelectable::GetCopyOfData(
                              const TTypeDescription* theType) const
```
Given a type description object, this method should return a copy of the data described by the selection. For example, in a text editor, if the selection is a range of characters, this method will return an object that contains those characters (a TText object). If called with NIL, return the data in your native type (If called with NIL, it means you are cutting and pasting between the same application or you are saving the old data for undo.).

```
void MSelectable::GetTypes(TDeque& theTypes) const
```
Fill in the supplied deque with a list of type description this selection is capable of publishing its data in. You are encouraged to publish in as many types as you can.

```
void MSelectable::ChooseType(const TDeque& theChoices,
                              TDeque& theChosenTypes) const
```
From the deque of choices, chose as many types as you'd like to receive data in for this selection. If some of the choices require the use of a shared library, it is advisable to make more than one choice and save a "fall back" position in case the shared library is currently unavailable (the user took the document home).

```
TDocumentSelection* MSelectable::AcceptData(const TDeque& theData)
```
Replace the data specified by this with the passed in deque of data. Each entry in the deque is a TTypeDescription*, MCollectible* pair encapsulated in a TAssoc*. The application owns the storage associated with the MCollectible* only.

```
TDocumentSelection* MSelectable::AcceptData(MCollectible* theData)
```
Replace the data specified by this with the passed in data. The type of the data is guaranteed to be in one of your application's native types. You can use GetClassNameAsString or GetClassNameAsToken to determine the actual type. The application owns the storage associated with the MCollectible*.

```
MCollectible* MSelectable::GetSelectionDataForUndo() const
```
This routine defaults to calling GetCopyOfData(NIL). Some applications will want to override this method to do smarter things to avoid copying.

```
MCollectible* MSelectable::GetSelectionDataForExchange(
                              const TTypeDescription* theType) const
```
This routine defaults to calling GetCopyOfData(theType). Some applications will want to override this method to do smarter things to avoid copying.

TDocumentSelection

The class, TDocumentSelection, should be used as a base class when applications are creating classes which represent the different kinds of selections supported in the application.

```
class TDocumentSelection : public MCollectible, public MSelectable {
  public:
    virtual  TAnchor*              Stickify() const;
    virtual  TDocumentSelection*   Duplicate() const;

  protected:
    TDocumentSelection();
};
```

```
TAnchor* TDocumentSelection::Stickify() const
```
Create a new anchor using the document selection specified in this as the specification for the anchor.

```
TDocumentSelection* TDocumentSelection::Duplicate() const
```
This routine provides a polymorphic clone capability to all selections. You must override this method until the Utility Classes provide similar functionality. The typical way to override this method is:

```
return new TMyType(*this);
```

Anchors & Links

Anchors are typically "sticky" document selections. By sticky, we mean that the anchor is resistent to editing changes in a document. When creating their own kinds of anchors, applications must use great care to insure that the information encapsulated in their subclass is enough to find the document selection independent of any change to the document. For example, a document selection in a text document is typically a range of characters. An anchor representing that selection needs more information because that range will typically be invalid immediately after an editting change. One possible representation would be an index into a range table managed by the application and saved with the rest of the data in the document.

Links are connections between two anchors. Operations on links include creating them, removing them, "following" them, pushing data from one sticky selection to the other, or pulling data.

TBaseAnchor

The TBaseAnchor class provides an abstract baseclass from which TSurrogateAnchor and TAnchor derive. It encapsulates what entity owns this anchor and a unique identifier for the anchor. These unique identifiers are assigned by the system and should not be needed by individual applications.

```
class TBaseAnchor : public MCollectible {
 public:
    virtual  unsigned long      GetUniqueID() const;
    virtual  void               SetUniqueID(unsigned long uid);
    virtual  const TEntityID&   GetEntityID() const;
    virtual  void               SetEntityID(TEntityID&);

 protected:
    TBaseAnchor(const TEntityID&);
};
```

TSurrogateAnchor

The `TSurrogateAnchor` provides the class which represents a surrogate for an actual anchor in the system. `TSurrogateAnchors` are relatively cheap to pass around and can be turned into a real anchor using a method of `TModel` (assuming, of course, that this is a surrogate for an anchor in your model. If not, using the anchor as a surrogate is the only way to use it).

```
class TSurrogateAnchor : public TBaseAnchor {
 public:
    TSurrogateAnchor(const TBaseAnchor&);
    TSurrogateAnchor();
    virtual ~TSurrogateAnchor();
};
```

TAnchor

The TAnchor class is an abstract baseclass which defines the protocol all anchors follow. Anchors contain a TDocumentSelection and are MSelectable objects. All methods of MSelectable are delegated to the embedded TDocumentSelection. They can be overridden if this is not the proper behavior. Subclasses of TAnchor should override methods to implement specific behavior for their kind of TAnchor. Subclasses should contain whatever information is necessary to find the appropriate "selection" independent of any editing changes to the document (or insure that the embedded document selection is resistent to editing changes).

```
class TAnchor : public TBaseAnchor, public MSelectable {
 public:
    virtual  void            Touch();
    virtual  const TTime&    GetModifyDate() const;
    virtual  void            SetModifyDate(const TTime& modifydate);

    virtual  void            SetDocumentSelection(const TDocumentSelection&);
    virtual  const TDocumentSelection& GetDocumentSelection() const;

    virtual  void            Follow(const THyperLink& theLink);
    virtual  void            Receive(const TDeque& theData);
    virtual  TAnchor*        Duplicate() const;
    virtual  void            ClearData();
 protected:
    TAnchor(
            const TEntityID&,
            const TTime& modifydate,
            const TDocumentSelection& theSelection);

};
```

`const TTime& TAnchor::GetModifyDate() const`
Return the modification date for the data specified by this anchor. This information is used by the semi-automatic update mechanism to send notification when clients have stale data [NOTE: This is not currently implemented].

`void TAnchor::SetModifyDate(const TTime& modifydate)`
Set the modification date of the anchor to `modifydate`. The modification date is the time when the data specified by the anchor was last changed.

`void TAnchor::Touch()`
Shorthand for `SetModifyData(TTime::Now());`

`void TAnchor::SetDocumentSelection(const TDocumentSelection&)`
Set the embedded document selection to the passed in selection. The old embedded selection is deleted and the passed in selection is copied.

`const TDocumentSelection& TAnchor::GetDocumentSelection() const`
Return a reference to the embedded selection.

`void TAnchor::Follow(const THyperLink& theLink)`
Override this method to implement your follow behavior. The TFollowedCommand will call this method on the followed anchor before posting the follow stimulus. The default method brings all of the document's presentations to the front using the following code:

```
TIterator* anIterator = GetModel()->GetPresentationIterator();
TPresentation* aPresentation = (TPresentation*) anIterator->First();
while (aPresentation != NIL)
{
    aPresentation->GetView()->GetLayer()->GetSystemLayer()->BringToFront();
    aPresentation = (TPresentation*) anIterator->Next();
}
delete anIterator;
```

`void TAnchor::Receive(const TDeque& theData)`
This method is called when data is pushed or pulled into the anchor. The default implementation is:

```
ClearData();
TDocumentSelection* retval = AcceptData(theData);
if (retval != NIL)
    delete retval;
```

`TAnchor* TAnchor::Duplicate() const`
This routine provides a polymorphic clone capability to all selections. You must override this method until the Utility Classes provide similar functionality. The typical way to override this method is:

```
return new TMyType(*this);
```

`void TAnchor::ClearData()`
This method is called by Receive when data is pushed or pulled into the anchor. This is part of the push/pull semantics (which are equivalent to select, copy, follow, select, clear, paste). The default implementation is:

```
fDocumentSelection->AcceptData(NIL);
```

THyperLink

The `THyperLink` class provides the representation for a link. Specialization is accomplished by subclassing `TAnchor`. There are no subclasses of `THyperLink`. The "`here`" anchor and the "there" anchor are just names for the two anchors in a link. The "`here`" anchor in a `THyperLink` is typically part of the current document and can be turned into a real anchor using a method of the `TModel`. The "`there`" anchor may be part of the current document or could belong to another document.

```
class THyperLink : public MCollectible {
 public:
    THyperLink(const TSurrogateAnchor& here,
               const TSurrogateAnchor& there, long uid = 0);

    virtual  const TSurrogateAnchor& GetHere() const;
    virtual  const TSurrogateAnchor& GetThere() const;
    virtual  unsigned long           GetUniqueID() const;
    virtual  void                    SetUniqueID(unsigned long);
};
```

Models, Documents, and Model Servers

Three classes in CHER represent a superset of the functionality available in MacApp. The main class that application writers should concern themselves with is TModel. Subclasses of TModel will contain all of the "document" data. TModel objects provide change notification to interested presentations. The TModel baseclass provides very little functionality; however, it does provide protocol which all models are expected to implement.

Unlike TModel, TModelServer provides a great deal of behavior for talking to other models and model servers. TModelServer subclasses act as a global controller to the model's data. It coordinates the application of command objects to all collaborators accessing the document. Only one method of TModelServer typically needs to be overridden.

TDocument subclasses act as a kind of local controller to the model's data. Again, very little overridding is necessary in TDocument.

TModel

The `TModel` class is the object to which command objects are applied. It contains all of the data in a document, all of the links and all of the anchors. The `TModel` class provides a change notification service so interested presentations are notified when changes occur to the model. The model defines protocol for saving and restoring all of the data associated with the model.

```
class TModel : public MCollectible {
 public:
    virtual  void            RegisterClient(
                                 TPresentation* whoToNotify,
                                 const TToken* whatChanged = NIL);
    virtual  void            CancelRegistration(
                                 TPresentation& whoToNotify,
                                 const TToken* whatChanged = NIL);
    virtual  void            Changed(TStimulus* changeStimulus);

    virtual  void            SetModelServer(TModelServer* theModel);
    virtual  TModelServer*   GetModelServer() const;

    virtual  void            AddPresentation(TPresentation* presentation);
    virtual  void            RemovePresentation(TPresentation* presentation);
    virtual  TIterator*      GetPresentationIterator() const;

    virtual  const TSet*     RetrieveLinks(TSurrogateAnchor& anAnchor);
    virtual  TAnchor*        LookupAnchor(TSurrogateAnchor& anAnchor);
    virtual  TIterator*      GetAnchorIterator() const;

    virtual  void            SetPendingClipboardData(MCollectible*);
    virtual  MCollectible*   GetPendingClipboardData(
                                 const TTypeDescription* theType = NIL);

    virtual  const TDocumentSelection*  GetDocumentSelection() const;
    virtual  void            SetDocumentSelection(
                                 const TDocumentSelection*);
    virtual  void            EstablishDocumentSelection(
                             const TDocumentSelection* newSelection,
                             const TDocumentSelection* oldSelection = NIL);
    virtual  void            Acquire();
    virtual  void            AcquireShared();
    virtual  void            Release();

};
```

```
void TModel::RegisterClient(TPresentation* whoToNotify,
                            const TToken* whatChanged)
```
Call this method to register a presentation to receive change notification. If the token is `NIL`, all changes will be notified. If the token is non-`NIL` then only notification stimuli which match the `whatChanged` token will be sent.

```
void TModel::CancelRegistration(TPresentation& whoToNotify,
                                const TToken* whatChanged)
```
Cancel change notification for a specific presentation. The `whatChanged` variable should match the `whatChanged` in a previous `RegisterClient`.

```
void TModel::Changed(TStimulus* changeStimulus)
```
Call this method to force change notification to occur. The change stimulus is propagated to all interested presentations via the normal responder/event mechanism. **The stimulus should be put on the heap. It will be managed by the system.**

```
void TModel::SetModelServer(TModelServer* theModel)
```
Set the model server to the passed in model server. This only works if there is a model server object in your address space.

```
TModelServer* TModel::GetModelServer() const
```
Return the model server object in your address space or `NIL` if there is none.

```
void TModel::AddPresentation(TPresentation* presentation)
```
Add a presentation to the list of presentations on this model.

```
void TModel::RemovePresentation(TPresentation* presentation)
```
Remove a presentation from the list of presentations on this model.

```
TIterator* TModel::GetPresentationIterator() const
```
Return an iterator which the application must manage. This iterator will iterate through all of the presentations.

```
const TSet* TModel::RetrieveLinks(TSurrogateAnchor& anAnchor)
```
Retrieve a set containing all of the links that contain anAnchor. The set is still owned by the model. NOTE: This is bad and will soon be fixed.

```
TAnchor* TModel::LookupAnchor(TSurrogateAnchor& anAnchor)
```
Given a `TSurrogateAnchor`, return the real `TAnchor` object that this surrogate anchor is a surrogate for. Of course, this only works for anchors in your own document. SurrogateAnchors in other documents will not be found and the call will return `NIL`.

```
TIterator* TModel::GetAnchorIterator() const
```
Return an iterator which the application must manage. This iterator will iterate through all of the anchors.

```
void TModel::SetPendingClipboardData(MCollectible*)
```
The current (**very temporary**) implementation of the clipboard uses two methods which you must override to store away the current contents of the clipboard during a cut or copy operation. The system will remember which document (model) contains the current clipboard data. Your `SetPendingClipboardData` routine should discard the old clipboard data and store the passed in data. It will always be in your native type. The corresponding Getter (below) will return the data in multiple types.

```
MCollectible* TModel::GetPendingClipboardData(
                        const TTypeDescription* theType)
```
Return the pending clipboard data (previously stashed away with `SetPendingClipboardData`) using `theType` as the return type.

```
const TDocumentSelection* TModel::GetDocumentSelection() const
```
Return what the model thinks the current document selection is.

```
void TModel::SetDocumentSelection(const TDocumentSelection*)
```
Set the document selection to a copy of the passed in selection.

```
void TModel::EstablishDocumentSelection(const TDocumentSelection* newOne,
                        const TDocumentSelection* oldSelection)
```
`EstablishDocumentSelection` is called by many command objects to set the selection after they are complete to establish the new selection (e.g. after a `TReplaceSelectionCommand`). The default code for `EstablishDocumentSelection` is:

```
 if (oldSelection == NIL)
    oldSelection = GetDocumentSelection();

 TDocumentSelection* newSelection = NIL;
 if (theSelection != NIL)
    newSelection = theSelection->Duplicate();

 TDualDocumentSelectionStimulus s =
        new TDualDocumentSelectionStimulus(kEstablishSelectionToken,
                                          newSelection, oldSelection);
 Changed(s);
 SetDocumentSelection(newSelection);
 delete newSelection;
```

```
void TModel::Acquire()
```
Acquire the model semaphore read/write. The document architecture does this whenever a command is being executed. Long running commands can copy the information they need; release the semaphore; operate on the data; then finally re-aquire the semaphore if they'd like to be nice citizens.

```
void TModel::AcquireShared()
```
Acquire the model semaphore read only. Trackers and presentations should acquire the model semaphore shared before trying to access the model. Of course, trackers and presentations should release the model semaphore when they are finished.

```
void TModel::Release()
```
Release the model semaphore.

TModelServer

TModelServer is an abstract superclass which implements much of the global behavior associated with applying commands to a document. It controls the collaboration (if there is one), it talks to the desktop manager and other system services to accomplish linking and data exchange. The model server object logs command objects to disk to add to the reliability of your program. There are many methods of the model server; however, only one is necessary to get started with CHER.

```
class TModelServer : public MServer, public MEventSource {
 protected:
    TModelServer(const char* theModel);
    virtual  TModel*           DoMakeModel(TModelServer* theModel) = 0;
```

```
};
```

TModel* TModelServer::DoMakeModel(TModelServer* theModel)
Override this method to make the kind of model associated with your document. The usual override for this method is:

```
    return new TMyModel();
```

TDocument

TDocument is an abstract superclass which acts as a kind of local controller to the model's data. The TDocument object applys commands to the model and assists presentations in finding the appropriate model for querying. Command objects are applied in a separate thread. This allows user interface stuff to be going on at the same time command objects are applied. Long running commands should explicitly release the model semaphore when appropriate (see TCommands).

```
class TDocument : public MResponder, public TModelClient {
 public:
    virtual  void              StartTheDocument();
    virtual  void              StartUp();
    virtual  Boolean           CleanUpBeforeQuit();

    virtual  void              SetCachedModel(TModel*);
    virtual  TModel*           GetCachedModel() const;
    virtual  void              SetQueryModel(TModel*);
    virtual  TModel*           GetQueryModel() const;
    virtual  const TEntityID*  GetEntityID() const;
    virtual  TModelServer*     GetModelServer() const;

 protected:
    TDocument(const TText& name, Boolean useLocalCache = FALSE);
    virtual  void              DoMakePresentations() = 0;
    virtual  TModel*           DoMakeCachedModel(const TEntityID&);
    virtual  MBaseTask*        FindModelServer(
                                  const TText& theModelServer,
                                  Boolean& startedInLocalAddressSpace) = 0;
    virtual  void              DoMakeQueryModel(const TEntityID&);
    virtual  void              SetModelServer(TModelServer* theServer);
             Boolean           GetUseLocalCache() const;
};
```

TDocument::TDocument(const TText& name, Boolean useLocalCache)
Create a new document. Methods that are overridden in your subclass of TDocument control the kind of model server created and the kind of model created. If you are collaborating and you are not the "collaboration server," (i.e. you don't have a model server on your machine) then set useLocalCache to true to force a local copy of the model on your machine. The name that is passed in is the "entity" name of the document.

void TDocument::StartTheDocument()
Most of the real work in document creation doesn't happen until this method is called. This is to get around the problem of calling virtual functions in constructors (thanks, Barney). The first thing that happens at startup is finding the model server. This is accomplished by calling FindModelServer. If there is to be a local cache, DoMakeCachedModel is called. Then DoMakeQueryModel is called. Finally, DoMakePresentations is called.

`TModel* TDocument::DoMakeCachedModel(const TEntityID& eid)`
The cached model is the model to apply command objects to in your address space. If the model server (and its associated model) are not in your address space (i.e. you are collaborating), you will probably want to have a cached model (If you don't have a cached model and you are collaborating, you must have a query model. See below). Typically, this is overridden to:

```
    return new TMyModel(eid);
```

`void TDocument::DoMakeQueryModel(const TEntityID&)`
The query model is the model your presentations will use when querying the model. Typically, this will be the model server's model if you are not collaborating or the cached model if you are collaborating. If these defaults are okay, then do not override this method. Alternatively, you could define a "query" model whichsupported the protocol of your normal model but communicated with the model server to fulfill any requests.

`void TDocument::DoMakePresentations()`
Override this method to create the presentations associated with the document. These presentations should register with the model for change notification.

```
MBaseTask* TDocument::FindModelServer(const TText& theModelServer,
                        Boolean& startedInLocalAddressSpace)
```
Find or start the model server. This method has no default implementation at this time but will soon. If you are not worried about collaborating, create a new model server, start it, set the startedInLocalAddressSpace to true, and, finally, return the model server.

`Boolean TDocument::CleanUpBeforeQuit()`
This method defaults to saving the document data (by issuing a SaveImage) at quit time. This will be more flushed out as the saving/versioning architecture is fully integrated into CHER.

`TModelServer* TDocument::GetModelServer() const`
Return the TModelServer* object associated with the document if it is in the same address space.

`void TDocument::SetModelServer(TModelServer* theServer)`
Set the TModelServer* object associated with the document if it is in the same address space.

Presentations

`TPresentations` encapsulate a model and a view. When a change occurs to model, all presentations registered with the model as interested in the change will receive change notification. This occurs by calling the `ModelChanged` method of `TPresentation`.

```
class TPresentation : public MCollectible {
 public:
    virtual  void        SetView(TView* theView);
    virtual  TView*      GetView() const;
    virtual  void        SetModel(TModel* theDocument);
    virtual  TModel*     GetModel() const;
    virtual  void        ModelChanged(
                            TModel* theModel,
                            TStimulus* theChange);
    virtual  MResponder* GetResponder(
                            TModel* theModel = NIL,
                            TStimulus* theChange = NIL);
 protected:
    TPresentation(TView* theView = NIL, TModel* theModel = NIL);
};
```

```
void TPresentation::ModelChanged(TModel* theModel, TStimulus* theChange)
```
When the model changes, this method is called. The default implementation for this method is:
```
  (GetResponder())->Respond(theChange, TRUE);
```

```
MResponder* TPresentation::GetResponder(TModel* theModel,
                                        TStimulus* theChange)
```
This is called by the default ModelChanged implementation to return what responder should be sent this notification. The default implementation of this method is:
```
    return GetView();
```

The CherApp

This class is a temporary class until CHER is integrated with the normal Pink Release.

```
class TCherApp : public TGraphicApplication {
 public:
    TCherApp(
            char* appName,
            MResponder *nextResponder = NIL,
            ApplicationState appState = kRunningState);
    virtual  void        SetDocument(TDocument*);
    virtual  TDocument*  GetDocument();
};
```

```
TCherApp::TCherApp(char* appName, MResponder *nextResponder,
            ApplicationState appState = kRunningState)
```
Create a new TCherApp. This constructor will be eliminated when CHER is part of the release and integrated with the user interface stuff.

Command Objects & Stimuli

Command objects encapsulate a user action and can change the document based on the user action. Command objects can be "done," "undone," and "redone." Command objects typically operate on a document selection which is part of the command object. The baseclass, `TCommand`, provides the protocol that all command objects respond to. Subclasses of `TCommand` override the `HandleDo`, `HandleRedo`, and `HandleUndo` methods.

CHER will provide the basic command objects for: cut; copy; paste; selection; starting links; completing links; following links; pushing data on a link; pulling data on a link; adding anchors to a document; and adding links to a document. It should not be necessary to override any of these command objects. Implementing the selection protocol will be enough to get all of these commands to work. The result of a command usually involves posting a stimulus to all of the interested presentations. The kind of stimulus posted by these command objects will be discussed with each command.

Many commands built into CHER have both a local effect and a global effect. For example, the cut command has the local effect of removing the current selection from the document. The cut command has the global effect of adding the current selection to the clipboard. Only the local effect of a command is undoable in the application that the command was done. The global effect is undoable in the global context (i.e. by going to the clipboard in this case). Almost all commands that developers will write themselves will only have a local effect. To implement this behavior, the `HandleDo` method of a command will do the global effect (with the help of the model server) and as the last statement of the `HandleDo` method do a `return TOtherCommand->HandleDo();`.

All command objects save the selection before the command is executed and restore the selection after an Undo by calling the model's EstablishSelection method.

TCommand

```
class TCommand : public MCollectible {
 public:
    enum  FlattenIntensity  { kEverything, kDoOnly } ;

 public:
    virtual  TCommand*      Do(TModel* theModel);
    virtual  TCommand*      Undo(TModel* theModel);
    virtual  TCommand*      Redo(TModel* theModel);

    virtual  void           SetDocument(TDocument*);
    virtual  TDocument*     GetDocument() const;

    virtual  void           SetModel(TModel*);
    virtual  TModel*        GetModel() const;

    virtual  void           SetDocumentSelection(const TDocumentSelection*);
    virtual  const TDocumentSelection* GetDocumentSelection() const;

    virtual  FlattenIntensity  GetFlattenIntensity() const;
    virtual  void              SetFlattenIntensity(FlattenIntensity);

    virtual  Boolean        ChangedByHandleDo();

 protected:
    TCommand(const TDocumentSelection&);

 public:
    virtual  TCommand*      HandleDo();
    virtual  TCommand*      HandleUndo();
    virtual  TCommand*      HandleRedo();
    virtual  Boolean        CommandIsUndoable();
};
```

`TCommand::TCommand(const TDocumentSelection&)`
Create a new TCommand object. The command object will eventually be applied to the passed in selection (if supplied) or the model's current selection (if it applies to a selection and none was supplied).

`TCommand* TCommand::Do(TModel* theModel)`
This method contains the machinery for "doing" the command. Eventually it calls the HandleDo method which you should override.

`TCommand* TCommand::Undo(TModel* theModel)`
This method contains the machinery for "undoing" the command. Eventually it calls the HandleUndo method which you should override.

`TCommand* TCommand::Redo(TModel* theModel)`
This method contains the machinery for "redoing" the command. Eventually it calls the HandleRedo method which you should override.

`void TCommand::SetDocumentSelection(const TDocumentSelection*)`
Set the selection that this command object will apply to.

`const TDocumentSelection* TCommand::GetDocumentSelection() const`
Return the selection that this command object will apply to.

`void TCommand::SetFlattenIntensity(FlattenIntensity)`
CHER will log command objects to the disk to add reliability to your application in case of extraordinary failures. Logging these command objects only involves logging the "do" part of the command. The flatten intensity is set to `kDoOnly` when logging command objects and `kEverything` at other times. Your `operator>>=` routine should check the flatten intensity using `GetFlattenIntensity` when flattening. Your `operator<<=` routine should be able to expand an object flattened with any intensity.

`FlattenIntensity TCommand::GetFlattenIntensity() const`
Return the current flatten intensity.

`TCommand* TCommand::HandleDo()`
Override this method to implement what your command does when "doing." The ultimate result of your command will probably post a stimulus to your presentations. Typically return this from `HandleDo`. The exception is when you are "doing" this command using another command (e.g. `TPasteCommand returns new TReplaceSelection command`). Long running commands can release the model semaphore in this method. They should reaquire the semaphore before exiting the command.

`TCommand* TCommand::HandleUndo()`
Override this method to implement what your command does when "undoing." The ultimate result of your command will probably post a stimulus to your presentations. Long running commands can release the model semaphore in this method. They should reaquire the semaphore before exiting the command.

`TCommand* TCommand::HandleRedo()`
Override this method to implement what your command does when "redoing." The ultimate result of your command will probably post a stimulus to your presentations. Long running commands can release the model semaphore in this method. They should reaquire the semaphore before exiting the command.

`Boolean TCommand::CommandIsUndoable()`
Return TRUE (the default) if this command can be undone. All commands that change the document must be undoable (unless they change into other commands - TPasteCommand becomes TReplaceSelection. TPasteCommand is not undoable but TReplaceSelection is.)

`Boolean TCommand::ChangedByHandleDo()`
If your command object doesn't change state during HandleDo (other than saving undo information), then an optimization can be made during collaboration. If the command objects changes (or one of the embedded objects changes) then this should return TRUE.

TCommandGroup

A command group is used to group a set of command objects that should be viewed as an indivisible unit for the purpose of do, undo, and redo. An example of when a command group is created is after the user issues a complete link command. Eventually, this turns into a group of commands: `TNewAnchor` & `TNewLink`.

```
class TCommandGroup : public TCommand {
  public:
    TCommandGroup();
    virtual  void            AddFirst(TCommand*);
    virtual  void            AddLast(TCommand*);
```

```
    virtual   TCommand*          Remove(const TCommand&);
};
```

```
void TCommandGroup::AddFirst(TCommand*)
```
Add the passed in command as the first command in the group.

```
void TCommandGroup::AddLast(TCommand*)
```
Add the passed in command as the last command in the group.

```
TCommand* TCommandGroup::Remove(const TCommand&)
```
Remove the command that `IsEqual` to the passed in command.

TSelectCommand

The `TSelectCommand` should be issued when changing changing selections in the document if you want selections to be undoable. This is currently a user interface question. At the conclusion of `HandleDo`, `HandleRedo`, or `HandleUndo`, the `TSelectCommand` will call the model's method, `EstablishDocumentSelection` on the new selection.

```
class TSelectCommand : public TCommand {
 public:
    TSelectCommand(const TDocumentSelection& theSelection);
};
```

TCutCommand

The `TCutCommand` has the local effect of cutting the current selection out of the document. It has the global effect of adding something to the clipboard. The local effect is accomplished by turning the `TCutCommand` into a `TReplaceSelection` command.

```
class TCutCommand : public TCommand {
 public:
    TCutCommand(const TDocumentSelection&);
};
```

TCopyCommand

The `TCopyCommand` has no local effect. It has the global effect of putting something on the clipboard.

```
class TCopyCommand : public TCommand {
 public:
    TCopyCommand(const TDocumentSelection&);
};
```

TPasteCommand

The `TPasteCommand` replaces the current selection with the "top" of the clipboard. This is accomplished by turning the paste command into a `TReplaceSelection` command.

```
class TPasteCommand : public TCommand {
 public:
    TPasteCommand(const TDocumentSelection&);
```

```
};
```

TReplaceSelectionCommand

The TReplaceSelectionCommand replaces the data specified by a selection or anchor with data encapsulated in the command object. The command object contains a TDeque of TTypeDescription, MCollectible* pairs which should be used when replacing the selection's data. A TDocumentSelectionStimulus with the token "DocumentSelectionContentsChanged" is issued as a result of this command. After this stimulus is posted, the model method, EstablishDocumentSelection, is called if the act of replacing the selection caused a new selection to be created. You will typically never create a TReplaceSelectionCommand yourself. CHER creates this command object as a result of cut, paste, push, pull, etc.

```
class TReplaceSelectionCommand : public TCommand {
 public:
    TReplaceSelectionCommand(const TDocumentSelection&);
    TReplaceSelectionCommand(TSurrogateAnchor& theAnchor);

    virtual  TDeque&     GetData();
    virtual  void        AddData(
                             const TTypeDescription&,
                             MCollectible* theData);
    virtual  void        AddData(MCollectible* theData);
};
```

TNewAnchorCommand

The TNewAnchorCommand is issued whenever a new anchor is created. A TStartLinkCommand becomes a TNewAnchorCommand after the global effect of TStartLink is done. After the TNewAnchorCommand is done or redone, a TSurrogateAnchorStimulus with the token "AddAnchor" is posted. If the TNewAnchorCommand is undone, a TSurrogateAnchorStimulus with the token "RemoveAnchor" is posted to all interested presentations.

```
class TNewAnchorCommand : public TCommand {
 public:
    TNewAnchorCommand(TAnchor* anAnchor);
    virtual  TAnchor*    GetAnchor() const;
};
```

TNewLinkCommand

The TNewLinkCommand is issued whenever a new link is created. A TCompleteLinkCommand becomes a TCommandGroup with TNewLinkCommands embedded in it after the global effect of TCompleteLink is done. After the TNewLinkCommand is done or redone, a THyperLinkStimulus with the token "AddLink" is posted. If the TNewLinkCommand is undone, a THyperLinkStimulus with the token "RemoveLink" is posted to all interested presentations.

```
class TNewLinkCommand : public TCommand {
 public:
    TNewLinkCommand(const THyperLink& aLink);
    virtual  const THyperLink& GetLink() const;
```

```
};
```

TStartLinkCommand

The TStartLinkCommand has the global effect of putting a new anchor on the "link board" and the local effect of adding a new anchor to the document. The local effect is accomplished by CHER issuing a TNewAnchorCommand.

```
class TStartLinkCommand : public TCommand {
 public:
    TStartLinkCommand(const TDocumentSelection&);
    virtual  TAnchor*    GetAnchor() const;
};
```

TCompleteLinkCommand

The TCompleteLinkCommand has to do alot of work. It has the (possibly) non-local effect of posting a new link command to another document (the document which issued the start link command). It has the local effect of adding an anchor and a link to the current document. This is accomplished using the appropriate command objects (TNewAnchor and TNewLink).

```
class TCompleteLinkCommand : public TCommand {
 public:
    TCompleteLinkCommand(TAnchor* theAnchor, const TDocumentSelection&);
    TCompleteLinkCommand(TAnchor* theAnchor);
    virtual  TAnchor*        GetAnchor() const;
};
```

TPushDataCommand

The TPushDataCommand has the (possibly) non-local effect of posting a TReplaceSelection command to the destination anchor. All type negotiation is handled via the selection protocol.

```
class TPushDataCommand : public TCommand {
 public:
    TPushDataCommand(
            const TSurrogateAnchor& sourceAnchor,
            const TSurrogateAnchor& destinationAnchor);
    virtual  const TSurrogateAnchor& GetSourceAnchor() const;
    virtual  const TSurrogateAnchor& GetDestinationAnchor() const;
};
```

TTickleCommand

The TTickleCommand command could be called the pull command. It is a command object which tells the other anchor to push data to me. This is accomplished by forcing the other side of the link to issue a TPushDataCommand.

```
class TTickleCommand : public TCommand {
 public:
    TTickleCommand(
```

```
            const TSurrogateAnchor& sourceAnchor,
            const TSurrogateAnchor& reply);
        virtual  const TSurrogateAnchor& GetSourceAnchor() const;
        virtual  const TSurrogateAnchor& GetReplyAnchor() const;
};
```

TFollowCommand

The `TFollowCommand` will "follow" a link. This involves posting a `TFollowedCommand` to the document containing the other side of the link.

```
class TFollowCommand : public TCommand {
  public:
        TFollowCommand(const THyperLink& theLink);
        virtual  const THyperLink& GetLink() const;
};
```

TFollowedCommand

The `TFollowedCommand` is posted to the document containing the destination anchor in a link. The "there" side of the THyperLink embedded in the TFollowedCommand is the destination anchor. The Follow method of the destination anchor will be called. Override this method to implement the proper follow behavior (typically scroll the anchor and its selection into view). Finally, a THyperLinkStimulus with the "Follow" token will be posted.

```
class TFollowedCommand : public TCommand {
  public:
        TFollowedCommand(const THyperLink& theFollowedLink);
        virtual  const THyperLink& GetLink() const;
};
```

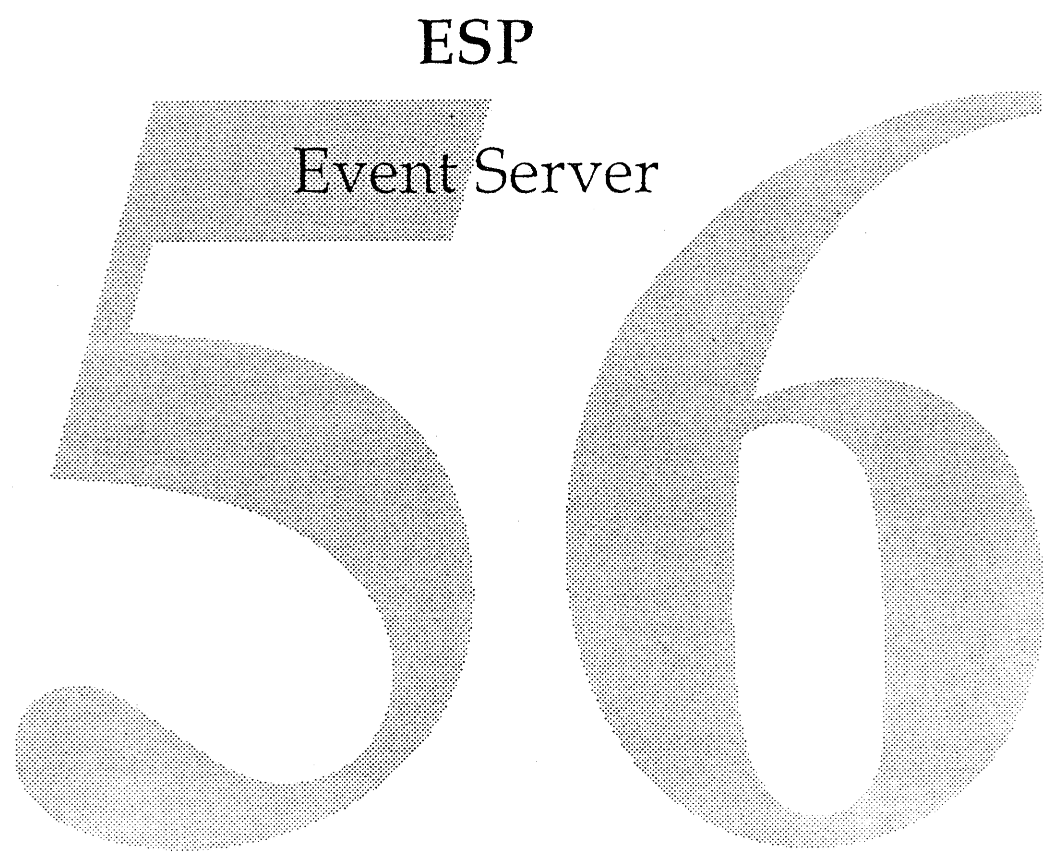
Stimuli

The stimulus that CHER posts are all named by the kind of objects encapsulated in the stimulus. The actual stimulus token posted at the end of a command is discussed with each command object.

```
class TSurrogateAnchorStimulus : public TStimulus {
  public:
        TSurrogateAnchorStimulus(
            const TToken& theStimulus,
            const TSurrogateAnchor& theAnchor,
            TModel* theModel);

        virtual  const TSurrogateAnchor& GetSurrogateAnchor() const;
        virtual  TModel*                 GetModel() const;
};

class THyperLinkStimulus : public TStimulus {
  public:
        THyperLinkStimulus(
            const TToken& theStimulus,
            const THyperLink&,
            TModel* theModel);
```

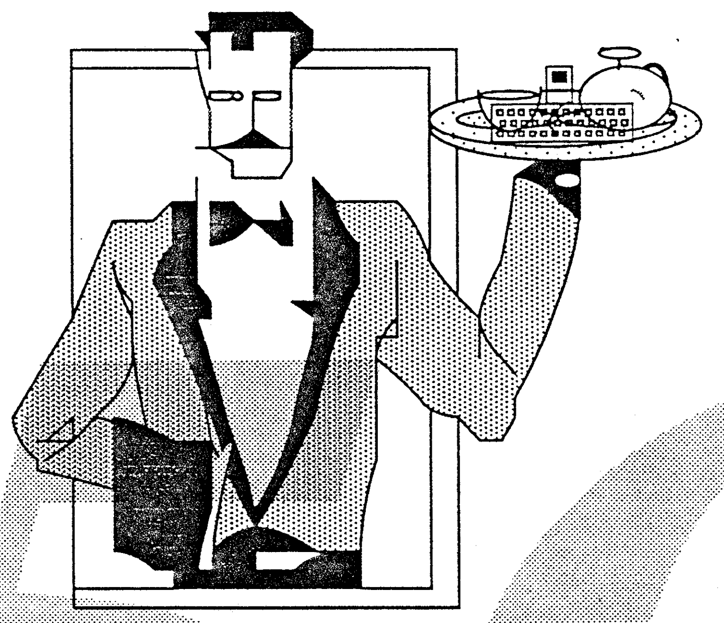
```
    virtual  const THyperLink&    GetHyperLink() const;
    virtual  TModel*              GetModel() const;
};

class TDocumentSelectionStimulus : public TStimulus {
 public:
    TDocumentSelectionStimulus(
            const TToken& theStimulus,
            const TDocumentSelection*);

    virtual  const TDocumentSelection* GetDocumentSelection() const;
};

class TDualDocumentSelectionStimulus : public TStimulus {
 public:
    TDualDocumentSelectionStimulus(
            const TToken& theStimulus,
            const TDocumentSelection*,
            const TDocumentSelection*);

    virtual  const TDocumentSelection* GetFirstDocumentSelection() const;
    virtual  const TDocumentSelection* GetSecondDocumentSelection() const;
};
```

ESP



Event Server

56



Event Server for Pink (ESP)

Arnold Schaeffer
x8117

The Event Server for Pink is responsible for routing events through the system. It must be able to deal with the usual mouse and keyboard events as well as provide good support for events generated by novel devices (Polhemus, data glove, etc.). The Event Server must not block, it must trust applications only as far as it can throw them, and it should be fast.

56

Introduction

The Event Server is primarily concerned with receiving events from the various devices attached to the system and distributing them using some well known protocol to applications in the system. There are a number of design goals for the Event Server presented here in no particular order. First, it must be reliable and it must not block because of a runaway application. It should not be possible for an ill-behaved application to take down the user's means of communicating with the system. Second, it should allow for events to be generated by previously unknown devices in a reasonable way. And, finally, it should be very easy for the developer of device drivers and applications to use. All of the external interface should hide any implementation details such as the use of Opus/2 messages, shared memory, and so on.

Overview

There are a small number of concepts that clients of the Event Server need to know. In theory, application developers will never see the interfaces presented in this document. The Pink Event and Message System is responsible for presenting events to the application. So, the clients of the information in this document are writers of device drivers, the implementor of the Pink Event and Message System, and adapter authors.

When the Event Server starts up, it registers itself with the name server so devices and applications in the system can find it. The Event Server instantiates a surrogate object for the Layer Server which is used for all communication with the Layer Server. Communication is necessary between the Event Server and the Layer Server to determine exactly which layer in the system an event needs to be dispatched to in the case of a positional event. Also, the Layer Server provides the Event Server with information about the frontmost layer.

Devices instantiate a surrogate object for the Event Server in their address space which is used for all communication with the Event Server. For the events generated by the mouse and keyboard, there is a predefined type of event called `TOSEvent`. For events which are generated by novel devices, a class `TOSExtendedEvent` has been provided.

1. Event Server starts up and registers itself with the Name Server so that devices and applications in the system can find it.
2. Event Server queries the Name Server to find the Layer Server. It will need to communicate with this dude later.
3. / 4. When devices start up, they register themselves with the Event Server.

Keyboard
Mouse
Name Server
Laser
Layer Server
Event Server

Figure 1 - What happens at startup time.

Once the Event Server has started up, it blocks on events to be posted by the devices attached to the system and then distributes these events to the proper applications. The Event Server does not believe that applications will be well behaved and thus takes necessary and adequate precautions against an evil application from taking over the machine.

When the user presses the mouse button (refer to Figure 2), an event is generated by the mouse driver and this event is sent to the Event Server using the `DistributeEvent` method. The Event Server receives the event, realizes that this event is a positional event and is to be distributed based on the application "under" the device. The Event Server queries the Layer Server to determine which task is

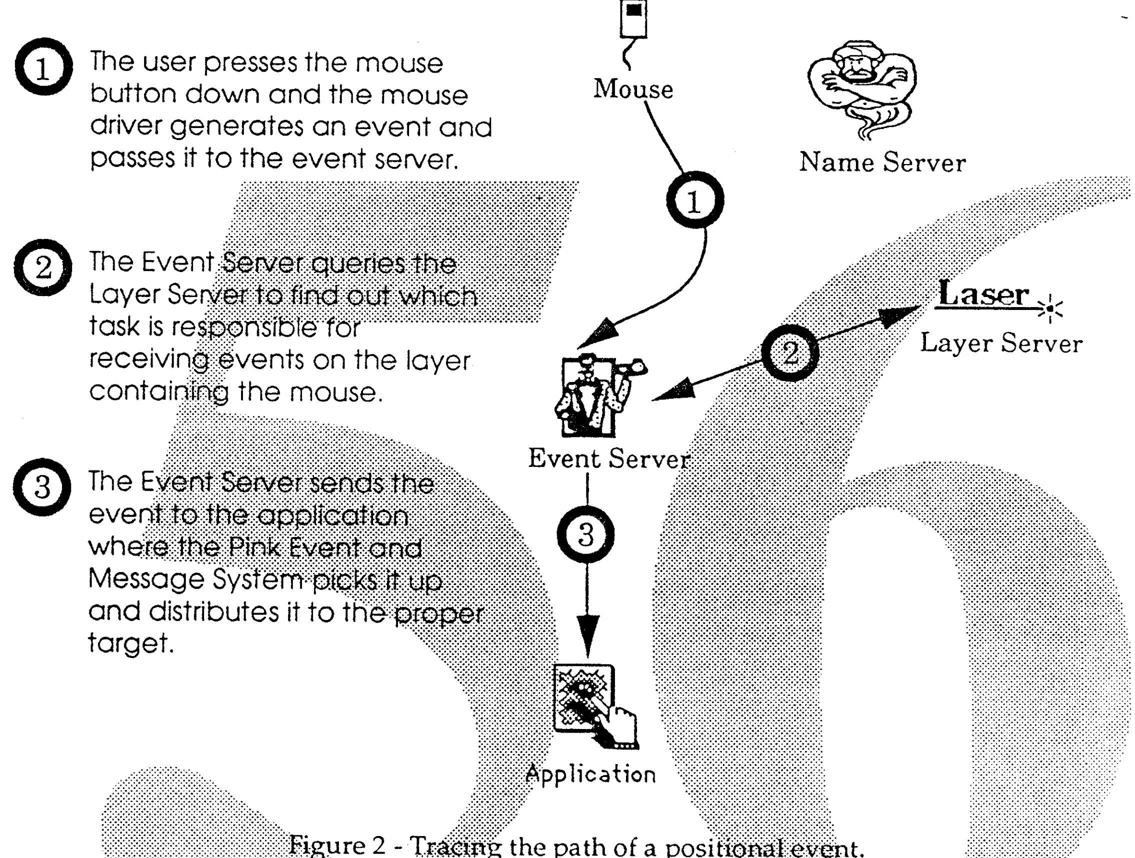


Figure 2 - Tracing the path of a positional event.

responsible for receiving events in this layer and then sends the event on to the application. The Pink Event and Message System is responsible for making sure that the proper `MResponder` object will receive the event.

In the case of a keyboard event (refer to Figure 3), the event is quickly passed onto the application because the surrogate LayerServer has cached the information concerning which application is the target of the keyboard.

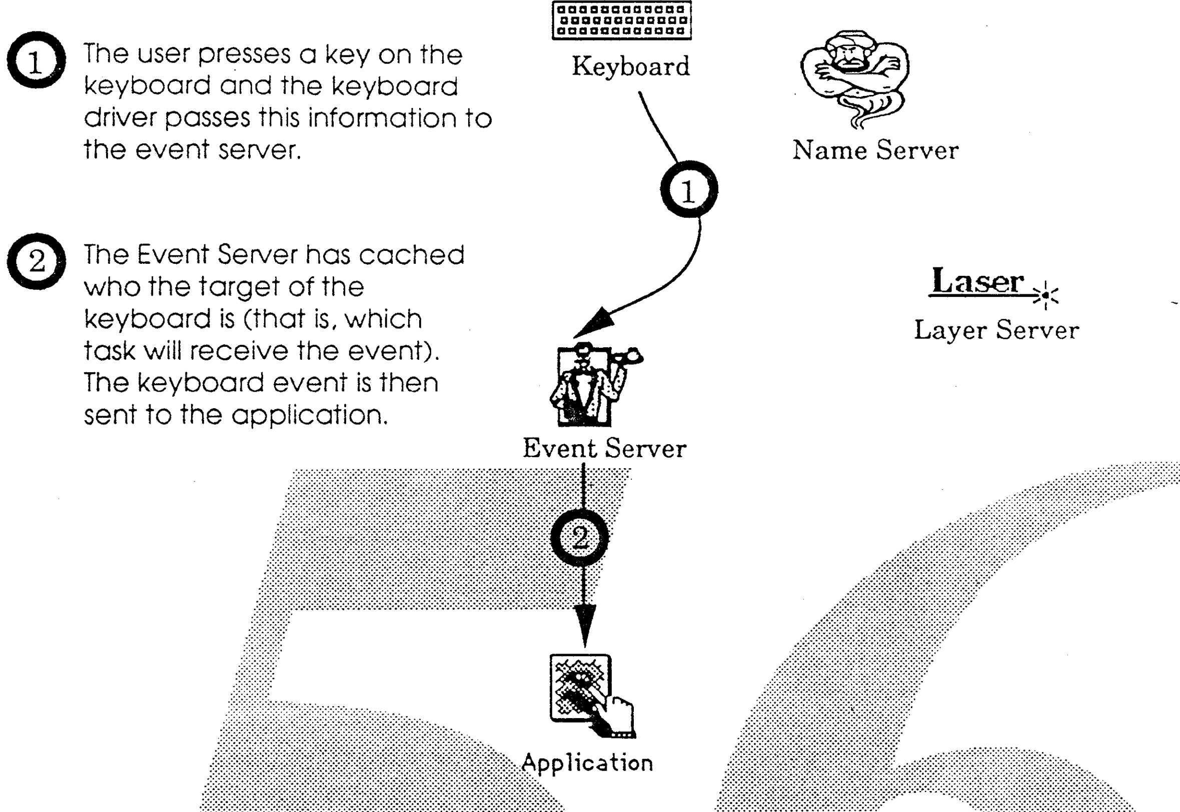


Figure 3 - Tracing the path of a nonpositional event.

Classes

The classes presented here are used internally by the application framework and by device drivers. Currently there is an architecture which supports a fixed size event and an extended event. This will be changing to a variable size event for all events. Clients of the application framework will never know this change occured. In fact, most of you reading this document should probably stop reading it now because unless you are implementing a device driver or the application framework, you are wasting your time

TEventServer

The `TEventServer` object is an abstract baseclass which encapsulates the protocol for talking to the real Event Server. The `TEventServer` object lives in your address space and talks to the real Event Server which lives in its own address space. This saves the application framework writer from having to know about the nasty details about Opus/2 IPC messages. It also insulates us so that future changes can easily be made to this protocol when porting the code to other architectures that allow messages of different sizes or different synchronization methods.

```
class TEventServer : public MServer, public MClient {
 public:
   virtual ~TEventServer();
   virtual void   Main(TMemory& startupInfo);

 protected:
   TEventServer(char* myprogramname,
```

```
        const TTaskSchedule& theExecutionCategory = kServerTask,
        size_t aStackSize = kDefaultStackSize);
  virtual void      NewEvent();
  virtual void      HandleNextEvent(TOSEvent*, TOSExtendedEvent*) = 0;
  virtual TOSEvent* AllocateTOSEvent();
};
```

`TEventServer::TEventServer(char*, const TTaskSchedule&, size_t aStackSize)`
Create the `TEventServer` object. Since this is an abstract superclass, the constructor can only be called by a subclass. You must override the `HandleNextEvent()` method to get this object to do anything. Remember, because a `TEventServer` is a `MTask`, your particular subclass must be started after it is created (by calling the virtual function `Start()`).

`TEventServer::~TEventServer()`
Destroy the `TEventServer` object.

`void TEventServer::Main(TMemory& startupInfo)`
Override this method if you have stuff to do that can't get done in the constructor but must get done when the `TEventServer` task starts up. The last thing you should do if you override this method is call the inherited `Main()`.

`void TEventServer::NewEvent()`
This method receives the next event from the EventServer and calls the `HandleNextEvent` routine which you need to override.

`void TEventServer::HandleNextEvent(TOSEvent*, TOSExtendedEvent*)`
Override this method to do whatever needs to be done when a new event comes in. The storage associated with the `TOSExtendedEvent` is yours to manage. Also, the storage associated with the `TOSEvent*` should be managed in a way consistant with how it was allocated in `AllocateTOSEvent`. See below.

`TOSEvent* TEventServer::AllocateTOSEvent()`
Each time an event comes in, a `TOSEvent*` needs to be used. The same one can be reused or you can create a new one each time. The default behavior is to always create a new one on the heap. You need to manage this in `HandleNextEvent` if you don't override this to reuse one `TOSEvent`.

MEventSource

All devices that will be generating events should be subclasses of `MEventSource`. Devices use these methods to register themselves with the `EventServer` and to post events.

```
class MEventSource : public MClient {
 public:
  virtual ~MEventSource();

 protected:
  MEventSource();
  virtual void RegisterDevice(DeviceClass, DeviceName,
                              Boolean async = FALSE);
  virtual void DistributeEvent(TOSEvent&, TOSExtendedEvent* e = NIL,
                              Boolean async = FALSE);
  virtual void PostEvent(TOSEvent&, TOSExtendedEvent* e = NIL,
```

```
                                 Boolean async = FALSE);
  virtual void PostEventToTask(const TSurrogateTask&, TOSEvent&,
                          TOSExtendedEvent* e = NIL, Boolean async = FALSE);
  virtual void PostEventToLayer(const TSurrogateLayer&, TOSEvent&,
                          TOSExtendedEvent* e = NIL, Boolean async = FALSE);
};
```

`MEventSource::MEventSource()`
Create a new `MEventSource` surrogate object. This object is used by the device driver for all communication with the Event Server.

`MEventSource::MEventSource()`
Destroy the `MEventSource` surrogate object.

```
void MEventSource::DistributeEvent(TOSEvent&, TOSExtendedEvent*,
                                   Boolean async)
```
Devices which want to have an event distributed positionally should call this method when they have an event to distribute. Note that this call blocks until the real Event Server has received the event unless the async flag is set.

```
void MEventSource::PostEvent(TOSEvent&, TOSExtendedEvent*,
                             Boolean async)
```
Devices which want to have an event posted nonpositionally should call this method when they have an event to post. Note that this call blocks until the real Event Server has received the event unless the async flag is set.

```
void MEventSource::PostEventToTask(const TSurrogateTask&, TOSEvent&,
                                   TOSExtendedEvent*, Boolean async)
```
Devices which want to have an event to a specific task should call this method when they have an event to post. Note that this call blocks until the task has received it unless the async flag is set.

```
void MEventSource::PostEventToLayer(const TSurrogateLayer&, TOSEvent&,
                                    TOSExtendedEvent*, Boolean async)
```
Devices which want to have an event posted to the task which receives events for a specific layer can use this call. Note that this call blocks until the event receiver task for the specific layer receives the event unless the async flag is set.

```
void MEventSource::RegisterDevice(DeviceClass, DeviceName)
```
This method should be called by all devices that wish to register themselves as generators of events. DeviceClass and DeviceName are both token ids. Use the token manager to get these. An example device class would be the token id for "Mouse." An example device name would be the token id for "Primary."

The TOSEvent Components

There are a number of classes (actually structs) which are used to represent the `TOSEvent`. Only the header is repeated here as the classes only provide methods for getting, setting, flattening and expanding.

```
// Typedefs
typedef TokenID       DeviceClass;
typedef TokenID       DeviceName;

// Temporary until we have the real time stuff
```

```
typedef HardwareTime  EventTime;
typedef short         VirtualKey;
typedef TokenID       WellKnownEvent;


class TModifier {
 private:
  unsigned char fBits;
 public:
  TModifier();
  void    SetOptionKey(Boolean on= TRUE);
  void    SetCapsLockKey(Boolean on= TRUE);
  void    SetShiftKey(Boolean on= TRUE);
  void    SetCommandKey(Boolean on= TRUE);
  void    SetControlKey(Boolean on= TRUE);
  Boolean GetOptionKey() const;
  Boolean GetCapsLockKey() const;
  Boolean GetShiftKey() const;
  Boolean GetCommandKey() const;
  Boolean GetControlKey() const;
};

class TWellKnownSideEffects {
 public:
  TWellKnownSideEffects();
  void     SetCausedLayerSwitch(Boolean on= TRUE);
  Boolean  GetCausedLayerSwitch() const;
  void     SetCausedNonPositionalFocusSwitch(Boolean on= TRUE);
  Boolean  GetCausedNonPositionalFocusSwitch() const;
  void     SetEventDistributedByXY(Boolean on= TRUE);
  Boolean  GetEventDistributedByXY() const;
  void     SetNeverSwitchLayers(Boolean on= TRUE);
  Boolean  GetNeverSwitchLayers() const;
};

class TOSEvent {
 public:
  TOSEvent();
  TOSEvent(WellKnownEvent event);
  TOSEvent(DeviceClass realDevice, DeviceName deviceName, TGPoint where,
          TModifier modifiers, WellKnownEvent event);
  TOSEvent(DeviceClass dclass, DeviceName dname, WellKnownEvent event);
  TOSEvent(DeviceClass realDevice, DeviceName deviceName,
           TModifier modifiers, VirtualKey key,WellKnownEvent event);
  TOSEvent(WellKnownEvent event, const TSurrogateLayer& newLayer,
           const TSurrogateLayer& oldLayer);
  TOSEvent(WellKnownEvent event, const TSurrogateLayer& newLayer);
  TOSEvent(DeviceClass dclass, DeviceName dname, WellKnownEvent event,
          LayerID aLayer);
  TOSEvent(WellKnownEvent event, LayerID aLayer);

  DeviceClass           GetActualDevice() const;
  DeviceClass           GetActingLike() const;
  DeviceName            GetDeviceName() const;
```

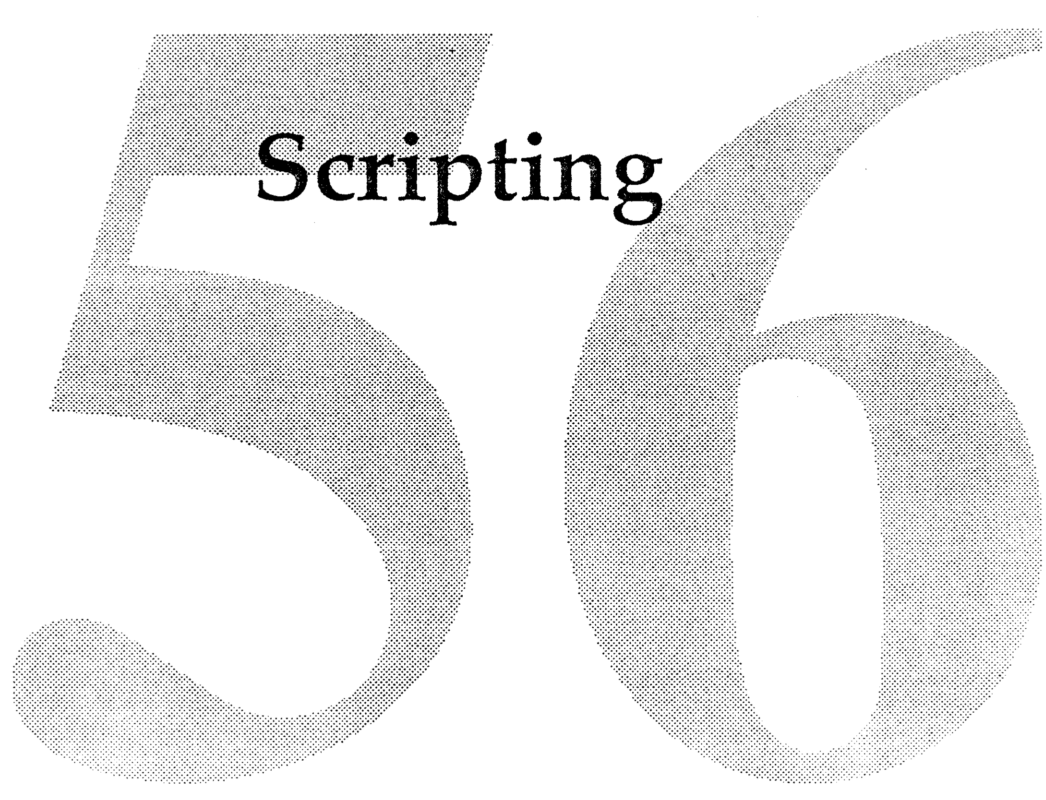
```
  EventTime               GetWhen() const;
  void                    GetWhen(TTime& aTime) const;
  TGPoint                 GetWhere() const;
  TModifier               GetModifiers() const;
  VirtualKey              GetKey() const;
  WellKnownEvent          GetEventType() const;
  TWellKnownSideEffects   GetSideEffects() const;;
  void                    GetNewSurrogateLayer(TSurrogateLayer&) const;
  void                    GetOldSurrogateLayer(TSurrogateLayer&) const;
  LayerID                 GetLayerID() const;
  void                    SetLayerID(LayerID);
  void                    SetActualDevice(DeviceClass dev);
  void                    SetActingLike(DeviceClass dev);
  void                    SetDeviceName(DeviceName dname);
  void                    SetWhen(EventTime when);
  void                    SetWhere(TGPoint& where);
  void                    SetModifiers(TModifier modifiers);
  void                    SetKey(VirtualKey key);
  void                    SetEventType(WellKnownEvent event);
  void                    SetSideEffects(TWellKnownSideEffects effects);

  TStream&                operator>>=(TStream&) const;
  TStream&                operator<<=(TStream&);
};
```

The TOSExtendedEvent Class

`TOSExtendedEvent` subclasses are used to represent events which can not be represented in a `TOSEvent` because they won't fit in an Opus/2 message. These events can be arbitrarily complex, use virtual functions, etc. Because of this flexibility, the event author (probably the person who writes the device driver) must provide implementations of the virtual functions for flattening and expanding extended events for transmission using the operating system interprocess communication mechanism. **Soon after d11, `TOSExtendedEvents` will be combined with `TOSEvent`. You will be able to subclass `TOSEvent` (or some other baseclass for events) to create your own kinds of events.**

```
class TOSExtendedEvent {
 public:
  TOSExtendedEvent();
  virtual ~TOSExtendedEvent();

  virtual TStream&   operator>>=(TStream& towhere) const;
  virtual TStream&   operator<<=(TStream& towhere);
};
```

56



Scripting

56

Introduction 1
  Purpose 1
  Objectives 1
  Strategy 2
Features 3
  Visual Scripting Language 3
  Automatic Task Recognition and Completion 3
  Explicit Scripting by Demonstration 5
  Effective Visual Metaphor 5
  Flexible Script Modification 9
Implementation Overview 10
  Scripts 10
  Scripting Server 12
  Script Application 15
  Task Recognition Engine 15
Application Scenarios 17
  Task Automation 17
  Interactive Examples 18
  Procedural Annotations 18
  Application Customization 18
Dependencies 19
  Selection Sets 19
  Command Objects 19
  End-User Objects 19
  User Interface 20

56

56

Introduction

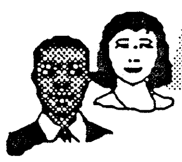
The Pink scripting system is designed to give end-users the ability to automate their complex or repetitive tasks in a way that is easy to use, universally available, and effective.

Purpose

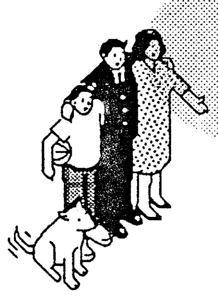
End-User Task Encapsulation

Repetitive actions are tedious. Complex actions are difficult to remember and are error-prone. Computers can be good at performing complex or repetitive tasks. However, they must be *programmed*. Unfortunately, this is not an option for most users.

A very small number of users are *application developers*. These people are usually professional programmers and are able to synthesize totally new applications for themselves or for distribution to others. Unfortunately, these people can never anticipate the needs of every potential end-user. In some cases, adding features or making the application more flexible in order to give it a better chance of meeting a diverse set of needs has the danger of making applications too complex [Norman 88].



Some users are *script writers*. These people create scripts or templates for themselves or for distribution to others. Frequently, they are part of a corporate information services group. On some systems they use batch files and shell scripts. On the Macintosh, they are able to use application-specific scripting or customization facilities such as those provided by MicroSoft Excel, 4th Dimension, or HyperCard (HyperTalk). Even though very useful, these systems cannot fully address the programming needs of end-users because few have the necessary skills or desire to write scripts. In the case of HyperTalk, most heavy users are programmers and regularly use other programming languages in their work [Nicol 89].



The largest group of computer users by far are not programmers and have no wish to become programmers. They are *end-users* who buy and use computers as a tool to enhance their work and personal productivity. These users must rely on others to anticipate their computer usage needs, an impossible task.

The primary purpose of the Pink scripting system is to provide these end-users a way to encapsulate their complex, repetitive, or procedural tasks. As such, it completes the set of programming systems necessary to satisfy the needs of the entire range of user types.

Objectives

The Pink Scripting System will be easy to use, universally available, and effective in its ability to automate end-user tasks.

Easy to Use

The Pink Scripting System will be easy and fun to use by end-users. It will be easy to enable, easy to disable, and easy to begin and control script execution. It

will be a tool that truly facilitates end-user tasks without getting in the way.

The Pink Scripting System will not require much additional skill or knowledge. Additional skills may be required, however, for advanced features such as the ability to modify or parametrize a script.

The Pink scripting system will employ a visual metaphor that is effective in distinguishing the script, its subject or task, and its parameters, and that facilitates editing, execution, and debugging.

Universally Available

The Pink Scripting System will be automatically available to users within and among all applications. It will require little additional effort for application developers to enable the base functionality. Enabling some advanced features may require additional developer effort.

Effective

The Pink Scripting System will correctly capture, display, and execute the user's intent. It will be as flexible as possible while remaining easy to use. It is desirable that this system be *Turing-complete*, but not at the expense of losing the end-user audience.

Strategy

The strategy for achieving these objects is to use the *language of the user interface* as the primary scripting language, to use a *scripting by demonstration* method of explicit and implicit script recording, and to employ an effective visual metaphor.

The remainder of this document describes the features and components of the Pink scripting system and gives some example applications of this technology.

Features

The Pink scripting system will feature a visual scripting language, automatic task recognition and completion, explicit scripting by demonstration, an effective visual metaphor, and flexible script modification.

Visual Scripting Language

One of the most important aspects of the Pink scripting system is the programming language used to record scripts. This language will determine, in large part, what programming constructs and operations are possible and how easy the system is to use.

Language of the User Interface

Externally, the Pink scripting system will use the *language of the user interface*. That is, users will develop scripts by performing the desired operation themselves using the normal set of objects, files, documents, applications, and commands available to them. The emphasis will be placed on *doing* rather than *telling*. Indeed, many users of this system will not be aware that they are "programming" at all. This method of visual programming is described as *programming by example* [Halbert 84], as *visual coaching* [Shu 88], and as *programming by demonstration* [Myers 88].

Using the language of the user interface makes scripting consistent with other user operations and builds upon the user's existing knowledge. It requires minimal extra learning by end-users. As such, it has the potential of reaching the widest possible audience.

This approach differs from the more traditional textual scripting languages in several important ways:

- It does not require the user to learn a new language. The language of the user interface employs familiar operations and leverages a user's experience so far in using the Macintosh.

- It does not require the user to learn new notations. For example, it would use the familiar icons for desktop objects rather than introducing a textual path name.

- It does not require abstract linear thinking. This, according to Norman Cousins, "...is the most difficult work in the entire range of human effort...". Scripting by demonstration gives users immediate visual feedback by employing familiar operations.

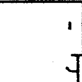
- It is extensible. As new applications and new operations become available to users, these operations may be used in scripts.

Automatic Task Recognition and Completion

The Pink scripting system will attempt to recognize repetitive tasks within an application as the user is performing them. As the user continues to perform the task, the system will give visual feedback to demonstrate that it is anticipating the user's actions. The user will then have the option of having the task completed automatically. This feature has been prototyped in HyperCard by

Allen Cypher [Cypher 89] and studied by the Apple Human Interface Group [Karimi 89].

Task Recognition



The Pink scripting system will constantly monitor the actions that are currently being performed by the user. As it does this, it will attempt to recognize a repetitive task that could be completed automatically.

This task recognition process must accommodate the following situations:[1]

- Mistakes. Users will make mistakes while performing a complex sequence of operations. They will either undo these mistakes or will in some way correct the errant operation. The scripting system must be able to disregard these mistakes as it is looking for repetitive command sequences.

- User action alternatives. Many applications offer users several different ways to accomplish the same thing. For example, in this word processor, one can make the next characters italic by typing "-i", by choosing the ***Italic*** option from the **Style** menu, or by selecting ***Italic*** in the **Style...** dialog box from the **Style** menu. Each of these user action alternatives cause the same effect and must therefore be considered equivalent during task recognition.

- Ordering. The scripting system must be able to detect equivalent but out-of-order sequences during task recognition.

- Extraneous behavior. Users may perform extraneous operations in order to confirm their actions that should not be part of the script. Examples might be opening a folder to see what is inside, playing a sound to make sure it is the one intended, and so on.

Anticipation

Once a repetitive task is recognized, subsequent user actions will be highlighted to show that the system is *anticipating* the actions. This feedback allows the user to gain confidence that the system understands the task being performed. If the user deviates from the anticipated action, the recognition process restarts.

A standard method of indicating an anticipated user action will be built into the system. This could be a color or some other form of highlighting. Users will also be able to look at the automatically-generated script while the system is anticipating user actions.

Task Completion

After the user is confident that the the system understands the repetitive task being performed, either through the anticipation feedback or by inspecting the proposed script, the user may request that the task be completed automatically.

Users will have the ability to control the speed of playback and display options. They will have the option to undo everything the system did, if necessary.

---

1. Many of these situations are research topics that have not yet been worked out. As such, this list represents ideals that may not be met in the initial implementation.

Explicit Scripting by Demonstration

In addition to automatic task recognition, the Pink scripting system will feature an explicit, user-initiated scripting capability.

The Pink scripting system will allow users to explicitly create a new script by performing a series of operations while the system records their actions. This series of actions, or script, will be stored for subsequent invocation or modification.

Users will have the ability to restart or disable the recording session. They will have the option to abort the recording and undo all operations performed since the recording began. This will help give users the confidence to try scripting and the tools necessary to restore the system state if they change their mind.

Effective Visual Metaphor

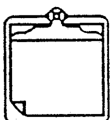
The Pink scripting system will feature a visual metaphor for inspecting and editing scripts. This visual metaphor has not yet been selected but it will include the following properties:

- Users must be able to quickly get a reasonable understanding of what the script does. The user objects that will be operated upon or are parameters to the script will be readily apparent. A complete understanding of the script may require closer examination.
- It must facilitate flexible editing operations, including cut, copy, and paste of discrete steps of the script.
- It must facilitate monitoring during script playback.

Using a visual representation in a programming-by-example system was explored by David C. Smith in *Pygmalion* [Smith 75]. He concluded that visual communication with a computer is a productive metaphor for assisting the thinking and learning processes of human beings More recent examples of visual programming systems are *The Fabrik Programming Environment* [Ludolph 88], which features a functional, bidirectional data-flow model, and *Prograph*, a very high-level pictorial object-oriented programming environment by The Gunakara Sun Systems. Both of these systems are available for the Macintosh.

The visual interface for the Pink scripting system has not yet been selected. Presented below are several possibilities: storyboards, films, and comics.

Storyboard Visual Metaphor



A story board is a collection of cards or pages, each describing one scene of a movie or play. It places the characters in the scene and describes their dialog or interaction.

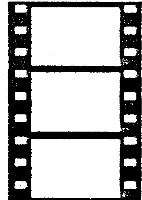
With a storyboard visual metaphor, each *frame* could represent a discrete step in a script. A set of steps in a script (i.e. a procedure or subprogram) could be represented by an *act*. The entire storyboard could represent the *screenplay* or *scenario*. A *program* could list the players (i.e. applications and objects) and give a synopsis of the play.

Each frame would have an area at the top that shows where the scene takes place and who the players are. At the bottom is the dialog, which could be used to explain what is happening or to specify details such as the file being operated

upon, etc.

During script execution, the scene currently being executed could scroll by.

Film Visual Metaphor



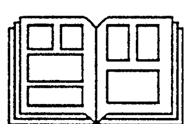
With a film visual metaphor, each *frame* could represent a discrete step in a script. A set of steps in a script (i.e. a procedure or subprogram) could be represented by a *scene*. The entire script could be represented by a *movie*.

Users could be shown several frames at once and would be able to browse through the scenes in order to get an understanding of what the script does. Script editing would be very much like splicing film.

During script execution the frame would scroll by, as in a real projector. All frames except the current one would be dimmed.

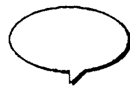
A film metaphor has the advantage that sound fits in very well, which could be used for special effects or explanations.

Comic Visual Metaphor



With a comic visual metaphor, each panel would represent a discrete step in a script. A set of steps in a script (i.e. a procedure or subprogram) could be represented by a *strip* or *page*. The entire script could be represented by a *book*.

During script execution, users would be shown a page at a time, which would represent the current procedure or subprogram. A special frame at the end of the strip or at the bottom of the page could be used to say what page to go to next, which might be useful to represent looping or branching.



Comics have an interesting feature that neither storyboards nor films have. The bubble provides a built-in notion of annotation, which may be helpful in explaining what is going on.

Comic books and comic strips are described in a wonderful book, *Comics and Sequential Art* [Eisner 85]. Some interesting techniques used in this medium are:

- Clocks in adjacent panels with different times indicate that something takes a long time.
- Panel sizes indicate relative importance or time requirements.
- Lettering style reflects the nature and emotion of the speech.
- A wavy-edged or scalloped panel border indicates past time. This might be useful to illustrate the environment or preconditions. Other border styles, such as a jagged edge, can indicate sound, emotion, or thought.

A comic strip construction tool called *The Comic Strip Factory* is available for the Macintosh from The Pacific BitWorks. This may be a useful tool for prototyping this type of user interface.

This option would be looked upon with glee by the Japanese and the young-at-heart but may turn off corporate America.

Object Visualization

The objects that are used or operated upon at each step of a script will have a visual representation or icon associated with them. These icons will be displayed to the user when inspecting and editing scripts.

The visual representation will be determined by the referenced object. The object will also be responsible for the granularity of the representation in order to reduce "icon overload." For example, a standard document icon might represent a character, word, paragraph, and chapter in addition to the entire document. Even so, these additional details will be available to the user as necessary.

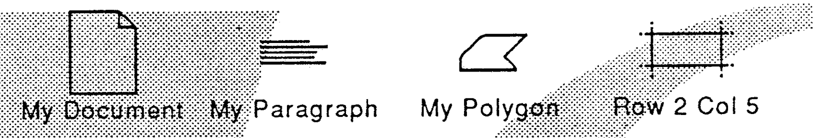


Figure 1: Several object icons[2]

Another object that may be referenced by a script is an *anchor* or sticky selection. Anchors in scripts are represented in the same way they are when editing documents.

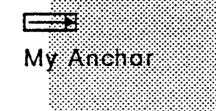
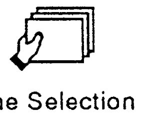


Figure 2: An anchor icon

Scripts may also reference the current selection. If one object is selected, the current script step would apply to that object. If more than one object is selected, the current script step would apply to each selected object, in turn.



The Selection

Figure 3: The "current selection" icon

Scripts may also reference a *selection set*. A selection set references other objects, either explicitly or through a user-specified pattern or filter. If the selection set contains one object, the current script step would apply to that object. If the selection set contains more than one object, the current script step would apply to each object, in turn.

2. These and subsequent icons are for the purposes of illustration only. The actual icons have not yet been determined.

Figure 4: A specific selection set document

Selection sets may also reference selection sets that must be supplied as a parameter when the script is invoked. Users (or perhaps previously-executed scripts) would be responsible for specifying objects that belong in this selection set before the script can continue execution. The user interface for specifying these objects will be the same as when constructing a selection set document. This unspecified selection set can be named in the script so that it may be referenced in more than one step of the script without ambiguity and so it can be distinguished from other selection sets.

Figure 5: A selection set has not yet been specified

Operation Visualization

It is also important to have a clear and unambiguous representation for the operations being performed at each step in a script. For example, the action of throwing away a desktop object might involve two nouns, *My Document* and *Trash*, and one verb, *move*. Internally, this is represented as a *command object*. This object will be responsible for determining the visual representation that will be used. An example is illustrated in Figure 6.

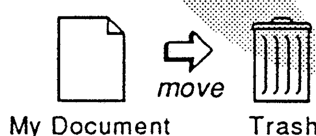


Figure 6: An example script step using noun/verb syntax

Figure 7 illustrates this operation using a declarative style: the same two nouns plus one preposition, *in*.

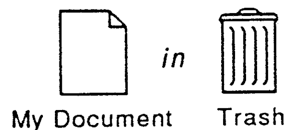


Figure 7: An example script step using noun/preposition syntax

The style of operation visualization that will be used in the Pink scripting system has yet to be determined. There is also a good potential for using animation to illustrate the script steps.

Flexible Script Modification

Users will have the ability to modify scripts once they have been recorded. This might be to correct mistakes made during the recording process or to further customize or specialize the script. There are several script modification operations that can be performed:

- Users may use cut, copy, and paste to reorder script steps, remove script steps, or add script steps.
- Users may partition several script steps into a sub-script in order to add structure or to increase clarity.
- Users may add script steps by selecting operations from a palette. These operations might include playback control operations in order to specify pauses, screen update controls, and so on.

Implementation Overview

The Pink scripting system implementation will include a script object to represent scripts and their behavior; a script server providing recording and playback capabilities; a script application that will support the script modification and playback features; and a task recognition engine to support the automatic task recognition and completion feature.

Scripts

There are two primary classes that are used to describe scripts: `TScriptStep` and `TScript`. The purpose of `TScriptStep` is to encapsulate the information necessary to describe an individual script step. The purpose of `TScript` is to collect these script steps and to provide an interface to the recording and playback capabilities of the scripting server (see page 12).

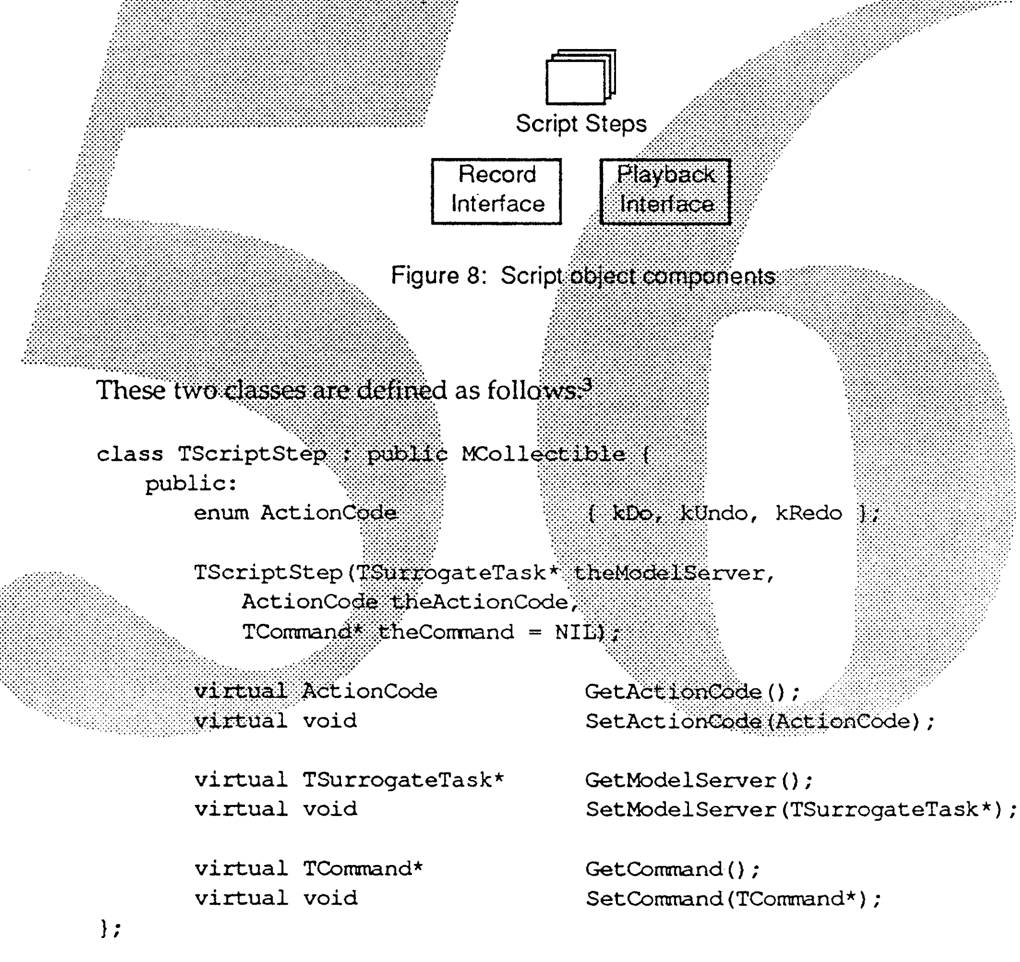


Figure 8: Script object components

These two classes are defined as follows:[3]

```
class TScriptStep : public MCollectible {
    public:
        enum ActionCode             { kDo, kUndo, kRedo };

        TScriptStep(TSurrogateTask* theModelServer,
            ActionCode theActionCode,
            TCommand* theCommand = NIL);

        virtual ActionCode          GetActionCode();
        virtual void                SetActionCode(ActionCode);

        virtual TSurrogateTask*     GetModelServer();
        virtual void                SetModelServer(TSurrogateTask*);

        virtual TCommand*           GetCommand();
        virtual void                SetCommand(TCommand*);
};
```

Script steps consist of an action code (`kDo`, `kUndo`, `kRedo`), a pointer to the applicable model server[4], and the command to perform in the case of a `kDo` action code.

3. This class definition and those that follow are abbreviated to include only important public member functions.
4. A model server provides a standard interface to documents see the *CHER* documents by Arnold Schaeffer and Larry Rosenstein. This will eventually be a `TEntityID` instead of a pointer to the actual model server.

```
class TScript : public MCollectible {
    public:
        virtual void                    Add(TScriptStep*);
        virtual TScriptStep*            Remove(TScriptStep*);
        virtual TIterator*              GetIterator() const;

        virtual void                    StartRecording();
        virtual void                    StopRecording();
        virtual Boolean                 NowRecording();
        virtual void                    RecordStep();

        virtual void                    Play();
        virtual void                    Play(TScriptStep&);

    protected:
        virtual TScriptStepReceiver* MakeScriptStepReceiver();
        virtual TScriptingClient*    MakeScriptingClient();
};
```

Scripts consist of a list of script steps and interfaces to the recording and playback capabilities of the scripting server. The process of manipulating the list of script steps and using the recording and playback capabilities are described in the following subsections.

`MakeScriptStepReceiver` and `MakeScriptingClient` simply return instances of class `TScriptStepReceiver` and TScriptingClient, respectively. These objects are used internally to receive script steps and to interact with the scripting server. These functions are provided for the convenience of subclassers.

Manipulating Script Steps

`GetIterator` returns an iterator appropriate for mapping over the list of steps. `Add` adds a script step to the end of a script. `Remove` removes the specified step from the script.[5] These member functions would be used by the scripting application, for example, when editing the script (see page 15).

```
// Example: Move the first script step to the end.

TScript         theScript;
...
TIterator*      anIterator = theScript.GetIterator();
TScriptStep*    aStep = (TScriptStep*) anIterator->First();
theScript.Remove(aStep);
theScript.Add(aStep);
```

Recording

The process of recording a sequence of commands involves creating a script object and using the recording-related member functions. `StartRecording` establishes a connection to the scripting server and starts receiving script steps as they are performed by the user (see page 12). `StopRecording` informs the scripting server to stop transmitting. `NowRecording` returns a boolean value indicating whether recording is currently in progress.

5. More flexible manipulation member functions may be necessary.

Internally, member function `RecordStep` is called when a new script step is received from the Scripting Server. The default behavior of this function is to simply call `Add`. This function is provided for the convenience of subclassers.

```
// Example: Create a script and start recording script steps.

TScript    theScript;
theScript.StartRecording();
...
theScript.StopRecording();
```

Playback

Member function `Play` with no arguments causes the entire script to be executed or played back. `Play` with a specific script step argument causes only that step to be played back. These functions establish a connection to the Scripting Server and sends it the steps to execute.

```
// Example:  Play an entire script.

TScript       theScript;
...
theScript.Play();


// Example:  Play the first script step.

TScript       theScript;
...
TIterator*    anIterator = theScript.GetIterator();
TScriptStep*  aStep = (TScriptStep*) anIterator->First();
theScript.Play(*aStep);
```

Scripting Server

The Scripting Server provides two major capabilities: a script recording engine and a script playback engine. The script recording engine receives commands from a model server and retransmits them as script steps to other interested servers, such as a script object or another server such as the Task Recognition Engine.

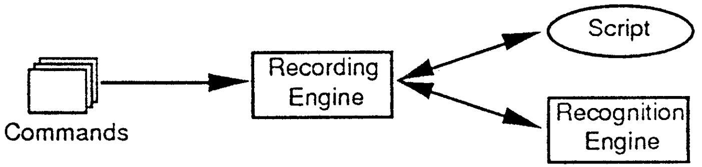


Figure 9: Script recording process

The script playback engine facilitates script playback by sending commands to the corresponding model server(s).

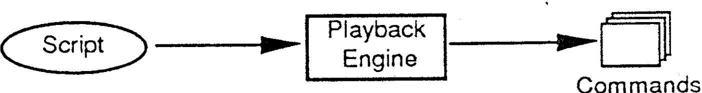


Figure 10: Script playback process

Class `TScriptingClient` provides the primary interface to the recording and playback capabilities of the scripting server. This is lower-level interface than that provided by `TScript`, allowing direct communication with the scripting server.

```
class TScriptingClient : public MClient {
    public:
        virtual void                StartTransmitting(TSurrogateTask&
                                        theScriptStepReceiver);
        virtual void                StopTransmitting(TSurrogateTask&
                                        theScriptStepReceiver);

        virtual void                Play(TScript&);
        virtual void                Play(TScriptStep&);
};
```

A second class, TScriptStepReceiver, is used to receive script steps sent by the recording engine.

```
class TScriptStepReceiver : public MServer {
    public:
        TScriptStepReceiver(TScript* theScript = NIL);

        virtual void                Main(TMemory& startupInfo);
        virtual void                ReceiveStep(TScriptStep*);
};
```

This class receives script steps from the scripting server as the user performs them. `RecieveStep` is called with each script step, in turn. The default behavior of this function is to call `RecordStep` for the script that was specified on the constructor, if any. Subclassers may wish to override `ReceiveStep` to do something else.

The scripting server uses an instance of class `TScriptStepTransmitter` internally to transmit script steps to a script step receiver.

```
class TScriptStepTransmitter : public MClient {
    public:
        TScriptStepTransmitter(TSurrogateTask* theScriptStepReceiver);

        virtual void                Transmit(TScriptStep&);
};
```

The constructor establishes a connection to the specified script step receiver and `Transmit` sends the specified script step.

Receiving Script Steps

Class `TScriptStepReceiver` is used to ask the scripting manager to start sending script steps as they are performed by the user. This class calls function `MakeScriptingClient` in its constructor to establish a connection to the scripting server and calls `function StartTransmitting` to initiate the transmitting of script steps. Function `ReceiveStep` is called to receive each script step the scripting server transmits. Function `StopTransmitting` is called in its destructor.

```
// Example:  Subclass TScriptStepReceiver to override member
// function ReceiveStep to do something interesting and start
// it receiving script steps from the scripting manager.

class MyScriptStepReceiver : public TScriptStepReceiver {
    public:
                                        TScriptStepReceiver();
        virtual void                    ReceiveStep(TScriptStep*);
};

MyScriptStepReceiver::ReceiveStep(TScriptStep* theStep)
{
    // something interesting
}
...
MyScriptStepReceiver  aScriptStepReceiver;
aScriptStepReceiver.Start();
```

Executing Script Steps

Scripts or script steps may be sent to the scripting server for playback with class `TScriptingClient`. `Play` with a script argument causes the scripting server to play the entire script. `Play` with a script step argument causes the scripting server to play only that step.

```
// Example:  Send a script to the scripting server for playback.

TScript           theScript;
TScriptingClient  theScriptingServer;
...
theScriptingServer.Play(theScript);


// Example:  Send a script step to the scripting server for playback.

TScript           theScript;
TScriptingClient  theScriptingServer;
...
TIterator*     anIterator = theScript.GetIterator();
TScriptStep*   aStep = (TScriptStep*) anIterator->First();
theScriptingServer.Play(*aStep);
```

Script Application

The purpose of the script application is to provide a user interface for recording, inspecting, editing, and playing back script objects.

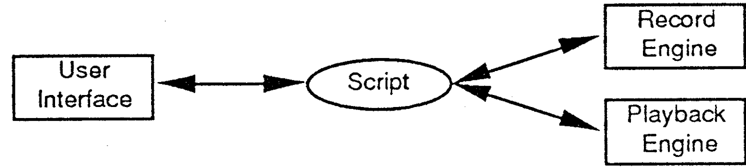


Figure 11: Script user interface

The script application will give users the ability to create new script objects and to start, stop, or cancel script recording.

The script application will provide an interface for viewing scripts, both during the recording process and for those that have been recorded previously. It is this interface the user will see when a script is opened.

The script application will have a flexible set of playback options. Included will be options to execute a single step, to execute the entire script without stopping, to display a message to the user and pause, to enable and disable screen updates, and to stop, restart, or terminate script execution. Each of these options are *scriptable* and may be incorporated into a script to provide flexible execution options. The ability to display and execute individual script steps, plus exception feedback from the script playback engine will provide an effective environment for script debugging.

The user interface for this application has yet to be determined but will be based on one of the visual metaphors proposed elsewhere in this document.

Task Recognition Engine

The purpose of the task recognition engine is to attempt to recognize repetitive tasks as the user is performing them. This is to support the automatic task recognition and completion feature (see page 3).

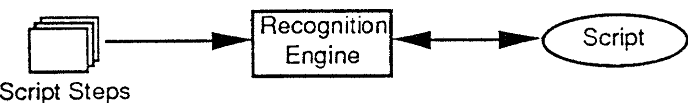


Figure 12: Task recognition process

The task recognition engine receives script steps from the scripting server. It compares this command object with recent command objects it has processed and attempts to detect a pattern. For example, applying the same sequence of commands to each document in a folder, or each card in a stack. This pattern of

command objects becomes a script that has been implicitly constructed.

When the task recognition engine detects a repetitive command object pattern, it will highlight the user actions that cause the next command object to occur. This feedback will demonstrate that the system is anticipating the next user action and has potentially constructed a script to perform the task.

The task recognition engine will include options to save the implicitly-generated script and to begin execution of the script in order to complete the user's task.

Application Scenarios

There are several applications or uses for this scripting technology. These include task automation, interactive examples, procedural annotations, and application customization. These are determined, in part, on how and where a particular script is made available to an end-user. These options include implicit scripts, document scripts, embedded scripts, annotation scripts, and bag-on-the-side scripts.

Task Automation

Implicit Scripts

Some scripts are generated implicitly by the scripting system and may be invoked by the end-user to complete a repetitive task (see page 3).

An example use of this is opening every document in a folder and using the global search and replace option to replace one phrase with another. After the first few documents, it might be nice for the system to take over and complete the task.

Document Scripts

Other scripts are explicitly recorded and may be saved as a document (see page 5). This document is simply another type of desktop object. It may be *opened* to inspect and edit the script or *launched* to invoke it.

An example use of a script document is to establish a specific arrangement of desktop objects. For example, one might have one arrangement of folders and other desktop objects for programming and another for writing a document.

Another example use of a script document is to "step through" a procedure, perhaps one that is infrequently-performed or requires user intervention and cannot be automated completely. Consider the task of a lawyer taking on a new client[6]:

- The script could first open a rolodex-like application, make a new card, ask the user to enter the appropriate information for the new client, and then pause.
- After the user indicates that the step is complete, the script could close the rolodex, open an invoice program, and ask the user to enter the billing rate for this client, and then pause.
- Other steps might include setting up file folders, printing a "welcome" letter with a personalized message, adding a reminder in an alarm program, and so on. Some of these steps the script could perform automatically and others it might pause while they are completed by the user.[7]

Another example use of a script document is to retrieve electronic mail messages from a remote system. The steps involved might be:

- Lauch a communication program
- Open a connection to the remote system

6. Thanks to Frank Leahy for this scenario.
7. It would be nice if the user could perform the steps in a random order, much like a check list.

- Log on
- Open a file to receive the email messages
- Send the commands necessary to retrieve the messages
- Log off
- Close the connection
- Quit the communication program

Interactive Examples

Embedded Scripts

Scripts may be embedded in a document to perform some action. This embedded script could be invoked on one of several conditions. For example, the script could be selected and invoked at the option of the user. Or, it could be invoked automatically when the portion of the document to which it is attached is visible.

An example use of this feature is in a help system. When the user asks for help on some particular operation, the help system could display a message explaining the option. The message could also contain a "show me" script that performs the operation as the user watches, pauses, and then automatically undoes any changes.

Procedural Annotations

Annotation Scripts

Scripts may also be used to provide procedural annotations. These scripts are similar to standard annotations but are executed instead of being incorporated directly.

An example use of this feature is when a reviewer proposes complicated changes to a document, such as rearranging several paragraphs or sections, changing fonts, and so on. The reviewer would start a script, make the changes, save the script as an annotation, and then undo the changes. The document's author could invoke the script to view the proposed changes interactively and decide whether to accept them. The author could even edit the script to remove an unwanted step, if desired.

Application Customization

Bag-on-the-side Scripts

Scripts may also be used to enhance or customize the features of an application. These scripts are similar to embedded scripts as described above, but are attached to an application menu or button instead of a document. This script might be defined to apply to the current selection.

An example use of this feature is to change a drawing tool to a pre-defined configuration in a paint program. The script could set the tool pattern, color, size, shape, enable the grid, disable mirroring, and so on.

Dependencies

There are several dependencies suggested by this system. Many of these dependencies regard selection sets, command objects, end-user objects, and user interface issues.

Selection Sets

Scripts must have some method of referring to a persistent set of selections. These selections may be anchors (sticky selections) or may be normal selections. The script will be applied to each selection in the set, in turn.[8] Some requirements for selection sets are:

- Users must have a way to specify the selections included in the selection set. This might be by making a selection, by dragging one or more selections into the selection set (anchors or normal selections), by specifying a pattern or filter from which a list of selections can be generated, or a combination of these techniques.
- Users must have a way to save selection sets for reuse, either in the script or in a selection set document.
- Users must have a way to indicate a selection set that must be specified by the end-user at script invocation time.

Command Objects

Command objects represent high-level application operations that result from user actions. Scripts include a sequence of command objects that are sent to documents when a script is invoked. Some requirements for command objects are:

- Command objects must include a visual representation, either in the form of an icon or a string. This will be displayed to the user when inspecting or editing a script.
- Commands objects must know what sequence of user actions causes the command to be invoked. This is to support anticipation (see page 4). One approach is to keep a record of the applicable end-user actions that caused the command object to be invoked.
- During the task recognition process (see page 4), identical commands that are being applied to different objects must be compared in order to determine the iteration pattern. One approach is to require command objects to provide comparison and iteration functions to the scripting system.

End-User Objects

End-user objects are things that end-users can manipulate. Included are desktop objects such as applications or documents, a paragraph of text within a document, a graphic in a drawing program, and so on. Some requirements for end-user objects are:

8. It is not yet clear whether the entire script or just the current script step (which might be an invocation of another script or sub-script) should be applied to each selection in the set.

56

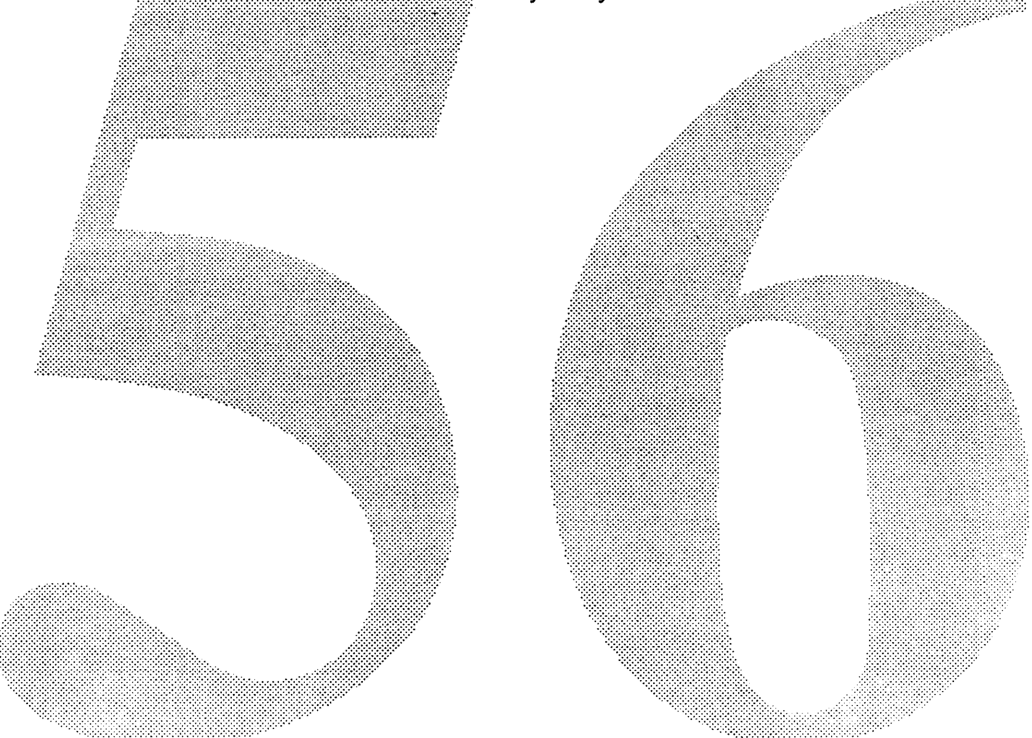
- End-user objects must know what command they support. The scripting system will use this information during script editing and to provide diagnostics.

User Interface

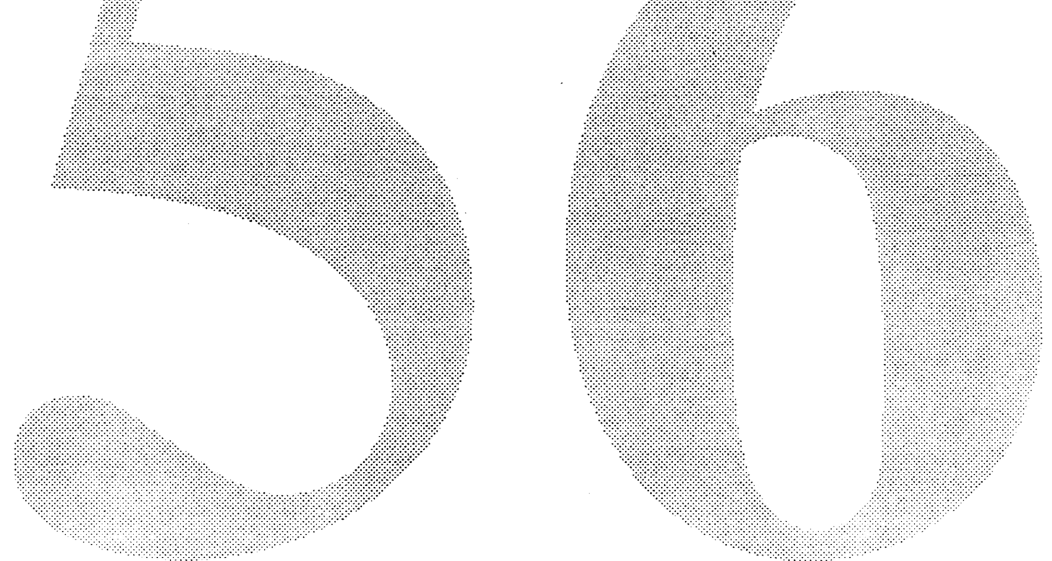
There must be a standard way of indicating an anticipated user action. This could be a color or some other form of highlighting.

During the script recording process, users could be asked to clarify their intent. For example, when they make a selection, the system could ask whether they meant that particular object, the object at that location, etc. The acceptable level of "intrusion" must be determined.

Bag-on-the-side scripts require that the user be able to attach a script to an application's user interface. This could be on a menu or a custom button. This method of user interface extensibility has yet to be determined.



Cypher 89	Allen Cypher, *Smart Macros*, Apple Computer, Inc., Intelligent Applications Group, May 1989.
Eisner 85	Wil Eisner, *Comics and Sequential Art*, Poorhouse Press, 1985.
Halbert 84	Daniel C. Halbert, *Programming by Example*, Xerox Corporation, Office Systems Division, Palo Alto, CA, Report No. OSD-T8402, December 1984.
Karimi 89	Shifteh Karimi, *Eager: A User Study*, Apple Computer, Inc., Human Interface Group, November 1989.
Ludolph 88	Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, Ken Doyle, *The Fabrik Programming Environment*, 1988 IEEE Workshop on Visual Languages, 1988.
Myers 88	Brad A. Myers, *Creating User Interfaces by Demonstration*, Academic Press, Inc., 1988.
Nicol 89	Anne Nicol, *Survey on User Programming*, The Human Interface Group, Apple Computer, Inc., July 1989.
Norman 88	Donald A. Norman, *The Psychology of Everyday Things*, Basic Books, Inc., 1988.
Shu 88	Nan C. Shu, *Visual Programming*, IBM Los Angles Scientific Center, Van Nostrand Reinhold Company, 1988.
Smith 75	David C. Smith, *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*, Ph. D. dissertation, Stanford University, 1975.



56

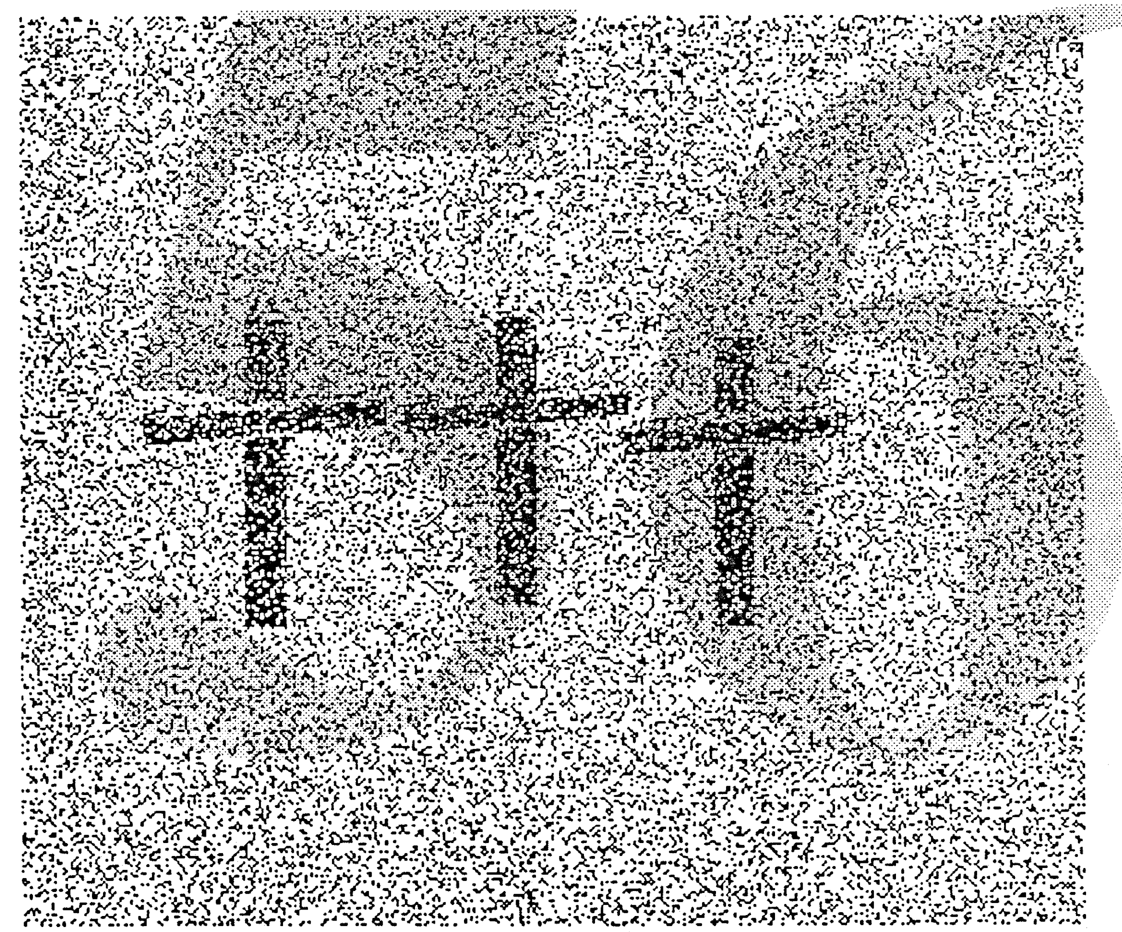
Donner Party

Collaboration Toolkit

56

Donner Party

Because collaboration isn't all
sweetness and light.



Sometimes it's dead serious.

Jack "Too Damn Fat" Palevich
x4-4738 AppleLink: PALEVICH1

56

What is Collaboration?

"**collaborate** — To work together, esp. in a joint intellectual effort."

— The American Heritage Dictionary

Collaboration is what people do when they work together to achieve a common goal. Most kinds of work require a great deal of collaboration. People's work days are made up of periods of individual work, interspersed with meetings, mail, phone calls, chance encounters, and other forms of collaboration.

Collaborative activities can be divided into two broad categories:

Real-time	People working together on the same thing at the same time.
Long-term	People working together on the same thing at different times.

When computers are added to the equation, there is an additional, orthogonal set of categories:

Centralized	All the work is done on the same computer.
Distributed	The work is done on several different computers.

Some example kinds of collaboration which will be possible in the Pink environment:

- Post-it Notes
- Classroom Administration
- Desktop Sharing
- Document Sharing
- Shared Whiteboard
- Group Decision Support System
- Network Administration
- Version control, undo
- Mail
- Personal AppleShare

To date personal computers have primarily enhanced people's individual work. Collaborative activities are still carried out pretty much by hand. For example, while people use word processors to write documents, the resulting documents are usually passed around in printed form. People scribble comments on the margins of the documents, then return them to the author. The author collates the comments, then laboriously edits them into the original document. The document is printed a second time, and the process repeats itself.

Pink is going to change this, because collaboration is built into the very fabric of Pink. Pink provides an environment in which people can collaborate with ease while retaining their privacy, their dignity, and their sanity.

Uses of Collaboration

Collaboration is something people already do. They just don't get much help from their computers. Pink is going to support both real-time and long-term collaboration.

Examples of real-time collaboration:

Example	Description
Remote access	A single user using a remote resources.
	Administering a remote file server.
	Editing a video tape using a remote editing suite.
Remote assistance	A user helping another user to do something. "Hey Jack, why can't I get TurboPinkDraw II to do this?"
	"Hmmm, let me look at you system. Oh, I see: just put 64BitAlbert in your System Folder—wait, I'll do it for you."
Micro meetings	"Hey Jack, what do you think of this diagram I just made?"
	"It's great—but what if we add this arrow."
	"Yeah, that's much clearer now."

Examples of long-term collaboration:

Example	Description
Document review comments	People type review comments into the original document. The author can accept or reject their suggested changes. It the author accepts their change, the system automatically updates the document.
HyperText links	Groups of documents can be hyperlinked together.
Source code sharing	Large software projects are easier to create & maintain because of automatic version control.[1]

---

[1] I am thinking of HOOPS here.

Why a Toolkit?

"Intuitions about collaborative work seem to be uniformly incorrect."

— Tom Erickson, Human Interface Review

If collaboration is such a great idea, why not just implement a universal collaboration application and be done with it? Well, unfortunately, there seem to be almost as many kinds of collaborative applications as there are people writing them. This is because people work together in so many different ways. Rather than force all forms of collaboration into a single mold, we provide a toolkit developers can use to write collaborative applications.

The Collaboration Toolkit consists of:

- a set of human interface guidelines for collaboration
- a document architecture which supports collaboration
- a graphics and event architecture which supports collaboration
- a collection of parts useful for building many different kinds of collaborative tools.

Having a common collaboration vocabulary shared by all applications makes it possible for users to shift smoothly from one kind of collaboration to another. Typical shifts are:

- Responding to a mail message by phoning the person. (The shift is from e-mail to telephone.)
- Leaving a voice mail message when the person you've called is away from their phone. (The shift is from telephone to voice-mail.)
- Sending a mail message to all the users of a particular file server. (The shift is from access control list to mailing list.)

How Collaboration is Built into Every Pink Application

Pink builds support for distributed collaboration into all of the code used to implement the Pink user interface. This means that every Pink application can, by default, be used in a collaboration between several people.

How is this done? It's done by modifying the traditional user-interface loop to include multiple users. The parts of the Pink system which enable this to happen are:

- The Document Architecture (Cher)
- The Event Server
- The Graphics Library (Albert)

This means that all documents, applications, and even the Desktop itself can be shared between several people at the same time. By making this kind of collaboration pervasive, we will be providing a better environment for our users.

56

Document Based Collaboration

The Pink environment is centered around the concept of documents. Users spend most of their computer time creating, editing, and reading documents.[2] We expect that most collaboration will center around documents as well. Because of this, we go to great lengths to support collaboration in the document architecture.

We support both real-time and long-term collaboration on documents. Real-time collaboration allows small groups[3] of people to simultaneously view and control a shared document. Long-term collaboration allows groups to create and manage a collection of shared documents. One of the goals of the collaboration toolbox is to provide a seamless transition between the two kinds of collaboration.

Desktop Based Collaboration

While documents are the meat of a user's computer diet, users spend a lot of computer time doing things besides editing documents.[4] We support non-document-based-work via a special kind of real-time collaboration called desktop sharing. In desktop sharing users give others access to their entire desktop. This allows the group to share essentially all applications, whether or not the applications use the Cher document architecture.

---

[2]When I wrote MeerKat, my Blue screen sharing program, I asked Apple engineers what applications they use. It turns out that engineers use text editors, program sh[illegible], and drawing programs. First level managers use presentation software and spreadsheets. Second-level mangers use AppleLink. (Of course everyone uses the Finder.)

[3]Two to twelve people. It takes two to tango, and twelve people make up a jury.

[4]Continuing the food analogy, the Finder is the bread & cereal group, desk accessories are [illegible] vegetables, and games could be the desert.

Desktop level collaboration gives the group access to the entire desktop of the shared machine. This is especially useful for the remote-assistance scenario, where one user is trying to debug another user's machine.

Real–time Collaboration

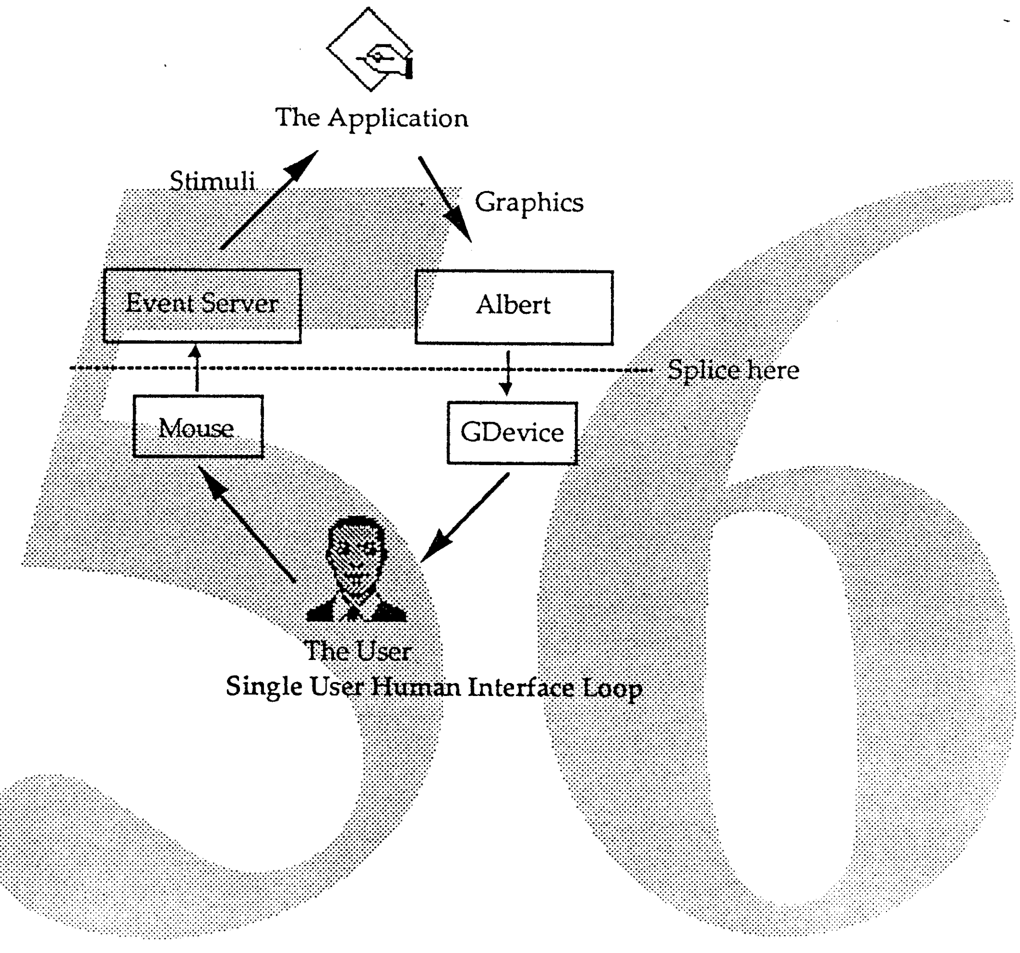
Real-time collaboration in Pink is based on the "what-you-see-is-what-I-see" model. This means that each collaborating user sees, as much as possible, the same display as all the others. This gives the group the illusion of actually sharing the same physical document.

Of course it is not always possible to attain this ideal. For example, consider the following case: One user has a color display and another user has a monochrome display. The two users are sharing the same document, and the document happens to have a color picture. The color picture will show up as a grey-scale picture on the monochrome screen. If the user on the color screen says "Look at the red flowers", the user with the monochrome screen will not be able to tell which flowers are red.

There are several ways the two users can work around their problem. One technique is to have the user with the color screen select the red flowers. The user on the monochrome screen will see which flowers are selected. Another technique would be to avoid using capabilities which are not present on all of the machines. In this case, since color is not available on all of the machines, the picture could be displayed in grays on both the color and the monochrome machine.

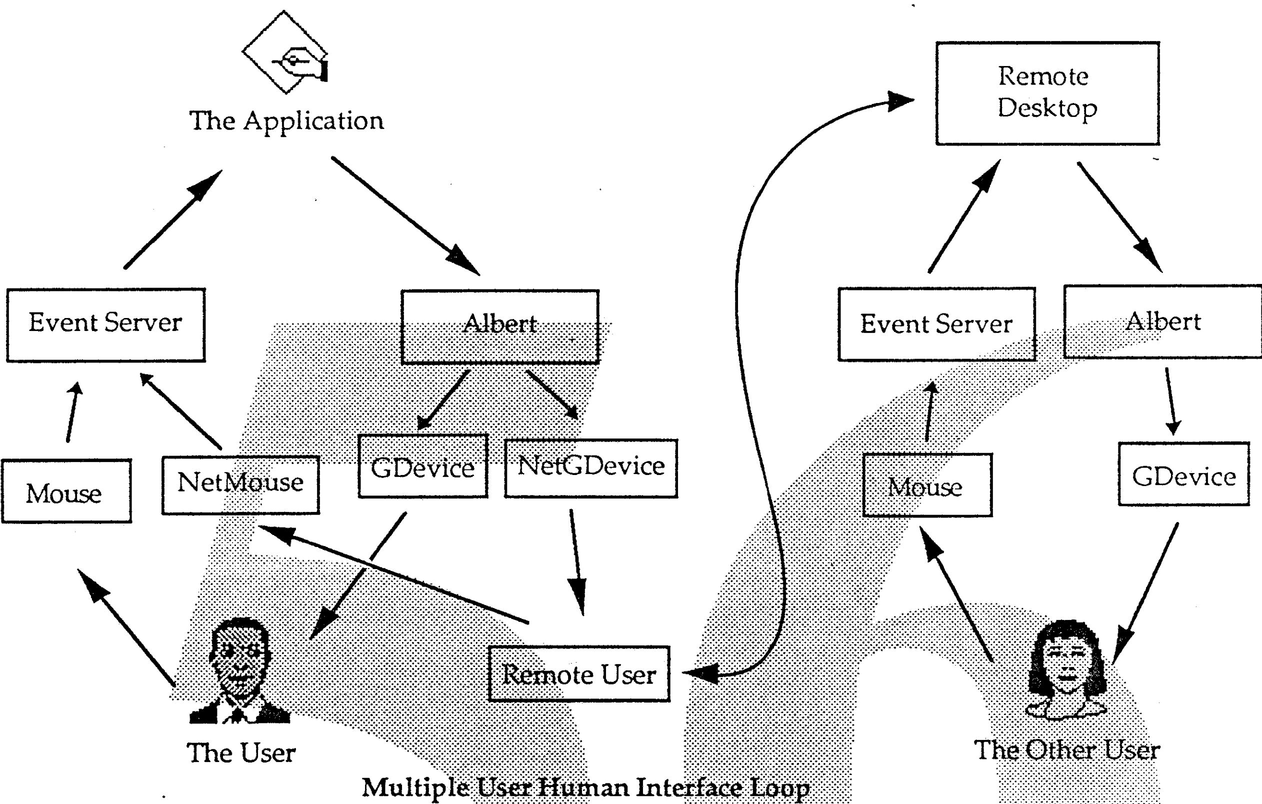
Desktop Sharing

In order to understand how desktop sharing is implemented, it is necessary to understand the human interface loop. This loop is what binds the users and the applications together. The loop begins when the application makes drawing calls to Albert. Albert tells each of its GDevices to draw. The GDevices then draw in their associated frame buffers. The user sees the display, and moves the mouse. The mouse notifies the event server. The event server sends a stimuli to the application, and the loop is complete.



Single User Human Interface Loop

In order to allow multiple users to access the same application, we have to splice into the human interface loop. By splicing into the loop at the level of the dotted line, we can provide desktop sharing without affecting the application at all, and with only minor modifications to Albert and the Event Server:



Multiple User Human Interface Loop

This is the way Macintosh screen-sharing programs[5] work today. In addition to graphics and mouse commands, the following items must also be sent across the network:

- Keyboard events
- Sounds
- Application-specific fonts
- Changes in desktop depth, size, and shape

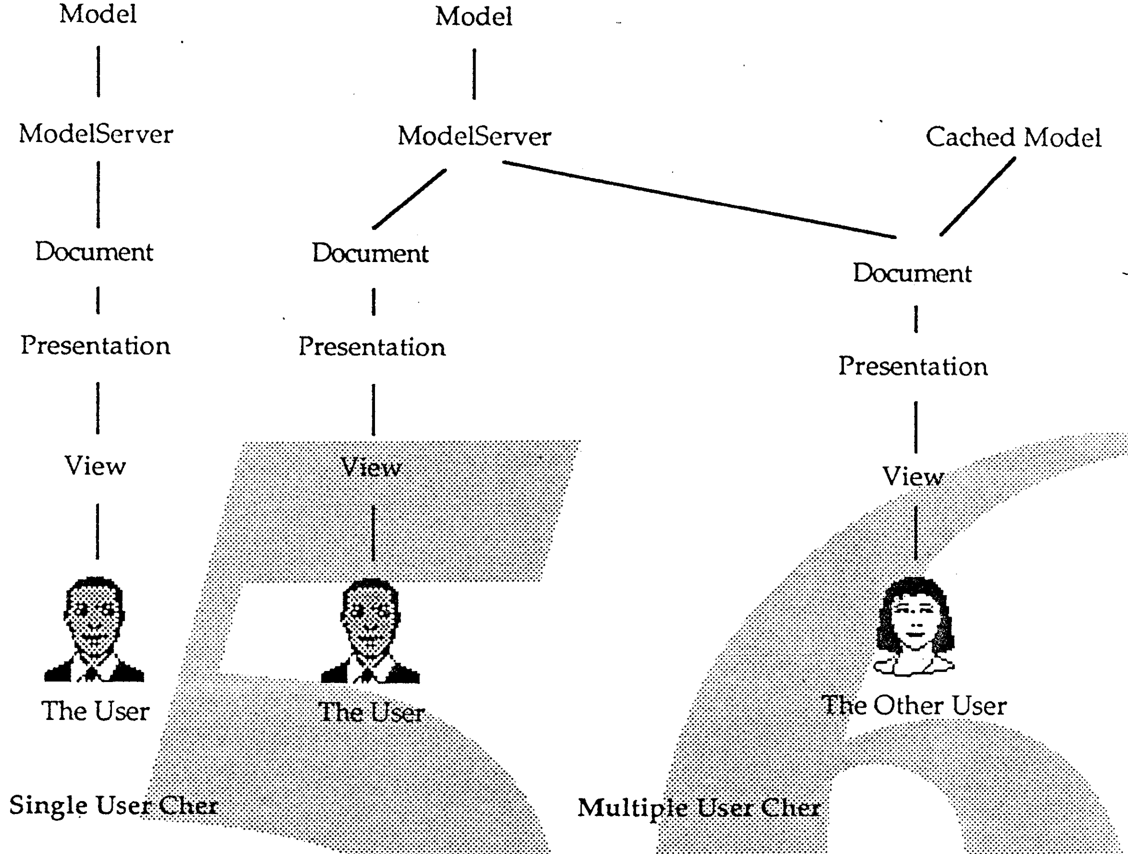
Document Sharing

When the application is written to use Cher, it can be used in real–time document-based collaboration. This is because Cher cleanly separates the data (called the Model) from the commands which modify the data. To share a document we run copies of the application on each user's machine. Each user's machine acquires a cached model, which contains a copy of the original data. When a user does something, their action is encapsulated into a command object. The command objects are sent to the model server. The model server distributes the command to all of the other collaborators. In this way all of the models are kept in sync.[6]

---

[5]like my MeerKat and Farallon's Timbuktu.

[6]See the Cher Document Architecture for more details.



Since Cher deals with changes to documents, it acts at the level of an undoable command. This means that Cher does not deal with tracking issues at all. If we want to provide distributed visual feedback of tracking, we will have to extend the current (d10) tracking mechanism.

Distributed Tracking

Tracking is what goes on when the user's input is echoed to the screen. Some common examples of tracking include:

- Defining a line by clicking and dragging
- Scrolling a window
- Typing text

Tracking is divided up into three phases:

1) The start, where tracking begins.
2) The middle, where tracking continues.
3) The finish, where tracking ends.

The middle phase lasts as long as the user is performing the action. It is during the middle phase that the rubber-banding of the line occurs.

To faithfully reproduce tracking across the network, it is necessary to transmit the start phase, as much of the middle phase as possible, and the finish phase. The middle phase can be transmitted asynchronously, using protocols which do not guarantee delivery of information. This is because lost parts of the middle phase do not affect the outcome of the tracking, and the remote users are usually more interested in viewing the current state of the tracking than its exact history.

The Whiteboard

By default, Pink applications are not aware that they are being controlled by multiple users. Whiteboard is an experimental Cher–based application which is directly aware of multiple users. Whiteboard is designed to assist small groups in their collaborative efforts. It acts as a cross between a white-board and a bulletin-board. A Whiteboard document is a shared space where users can jot down their ideas, leave clipboards and documents for one another, or engage in mini-meetings. It is similar in spirit to Andy Atkins' ShareBoard, Jay Fenton's Playground, and Gregg Foster's Cognoter[7].

The Whiteboard application will be implemented in stages. It will start out very small, and grow over time. The first generation whiteboard will be little more than a shared scrap-book. As editing functionality is added to it, it should become something like a shared MacDraw. Finally, it should end up as a prototype distributed component document.

[7]Cognoter was written at Xerox PARC.

A Collaboration Toolbox

This section describes the objects, classes and servers which support collaboration. Taken as a whole, this is the Collaboration Toolkit. The careful reader will note that many of the things described here are actually implemented by other parts of the Pink system.

Some of the following items are very concrete — they describe things which are already implemented. Other items are more abstract — they are place holders provided to stimulate discussion.[8]

Users as first class objects

The TPerson class gives Pink programs a way to deal with individuals. TPerson objects contain, or at least know how to find out, information about particular human beings.[9] This information includes things like:

- Name
- Face
- ID
- Interests, history
- Preferences / customizations
- Location
- Resources that they own
- Groups they belong to
- Where to send their mail

Once we start keeping information about people, all sorts of privacy issues arise. In general, we hope to provide appropriate access controls to allow information to be kept private.

Groups as first class objects

The TPeople class gives Pink programs a way to deal with groups of individuals. TPeople objects contain, or at least know how to find out, information about groups of human beings. Groups are like individuals in that they can receive mail and own things.

It might be useful to allow groups of other network objects, like printer pools or server pools.

Network Shareable Resources

Network shareable resources are things which can be shared over the network. These include hardware resources, like printers, and software resources, like data bases. Pink provides the basic technology to allow these resources to be shared. Hardware resources can be shared via the network-transparent client & server classes. Software resources can be shared either via servers or via copying.

---

[8] If something in this list doesn't jibe with your expectations, please tell me. Jack Palevich (x4-4738)

[9] It seems likely that the TPerson, TPeople, and TNetworkShareableObject classes will all be subclasses of the NetComm TIdentity class.

Some typical shareable hardware resources are:

- Computers
- Printers
- Video equipment
- Modems

Some typical shareable software resources are:

- Data bases
- File systems
- Fonts
- Applications

Butlers

In times past it was common for rich people to employ butlers to run their households. The stereotypical butler was a competent, self-effacing person who moved quietly behind the scenes, ensuring that everything ran smoothly. In today's business world the area associate often acts in a similar capacity.

A Butler acts on behalf of a user. A butler does the following things:

- Protects the user from the outside world by screening visitors, calls and mail messages.
- Acts as an answering machine for messages, mail, fax, and phone calls.
- Notifies user of outside requests
- Enforces user's policy on privacy and access control issues
- Guards the user's time when meetings are scheduled.

A user's Butlers appears when some outside event occurs.

During attempts to start a real-time collaboration, butlers can be employed to tell "white lies" about your availability.

Current personal computer systems do not provide butlers. As a result, the user is exposed to a barrage of rude and distracting messages.

The Address Book

Pink will have a way for both programs and users to find the objects used in collaboration. This will be something like today's Chooser, only much better. The exact details and capabilities are not yet determined. In general, the following kinds of ideas are being considered:

- The user-interface should look something like the one used to manipulate files & folders.
- It should be possible to browse the space of all entities.
- It should be possible to search for entities.
- Once an entity is found, it should be possible to cache an alias to the entity in a private address book.
- Interesting information will be associated with an entity. It will be possible to view this information and use it as a key for sorting and searching.

Administering Users, Resources & Groups

There must be a way of dealing with all of these entities. Happily, the Apple Mail group has made great progress in this area. We hope to use their naming protocol and their ADAS[10] servers as the basis for our system.

Authentication

It is necessary to provide a way for people (and their agents) to prove their identities. Again, the Apple Mail group has anticipated this need, and is providing authentication as part of their ADAS system.

Contact Logs

The Collaboration Toolkit will keep track of the network resources used during collaborations. It will be easy for users to browse and reconnect to recently used resources.

Issues

There are a number of issues which need to be resolved. Many of these are policy issues. We will have to conduct user tests, surveys, and other experiments to decide upon the right policies.

Sharing Resources

Resources are things, like documents, fonts, and applications, which are used during a collaboration. While the data in a document must be shared, it is necessary to ensure that all users have the right fonts and applications.

---

[10]Apple Distributed Authentication Server

It's easy to determine which resources are needed for a collaboration, and it's easy to check if each collaborator has the necessary resources. The issue is what do when some users lack the necessary resources.

One policy is to force users to provide their own copies of the needed resources.

Alternatively, it is possible to borrow the needed resources for the duration of the collaboration.[11] [12] Borrowing raises many questions:

- Can we ensure that it is truly a temporary borrow? (Guilt and its uses in user interface)
- Will Font vendors go for it?
- Will App vendors go for it?
- Can we support site-licensing ?
- Should we lend document viewers? What about document editors?
- Should we put a time limit on borrowing?
- Can things be borrowed for long–term collaboration, or just for the duration of a real–time collaboration?

Graceful Degradation

Borrowing solves the problem of sharing for resources which can be copied. We run into a harder problem when we consider hardware resources, such as monitors, which vary from user to user. What should we do when two users have different kinds of hardware?

One alternative is to provide the best possible service to each user. This should be the default. In some cases, though, this can lead to confusion, because different users will see different things. To reduce confusion, it should be possible to switch the collaboration into "greatest-common-denominator" mode, where the graphics are displayed the same way on every users machine.

---

[11] In the case of fonts, this is very similar to what is done when a document is printed on a printer: any fonts the printer doesn't already have are downloaded to the printer along with the document.

[12] The USA project investigated sending both instance variables and methods across networks. It will be interesting to see how applicable their work is to this problem.

Some strategies for working around differences in hardware are listed below:

- Virtual memory (although this could lead to thrashing)
- Scaling and/or scrolling when viewing a large screen from a small one
- Gray scale (or Black & white dithering) substituted for color
- Dropping frames from an animation
- Representative frame substituted for video
- Degradation of sound (Stereo to Mono, sampling rate reduction, delays)
- Placeholder objects (Rectangle which says "I am a pipe")
- Distributing the results of a computation rather than duplicating the computation on each machine.

Internationalization

Collaboration adds a new dimension to internationalization issues. This is because the individuals involved in a collaboration might be using different languages. It would be desirable to have error messages, dialogs, and menu text appear in each user's native language.

Desktop sharing also introduces the problem of trying to use a particular language system with a different language system's keyboard. This would happen if an American user tried to provide remote assistance to a Japanese user.

How fast does the network have to be?

Real-time collaboration is a great idea, but it only works if the data can be transmitted to all the collaborators quickly enough to provide the illusion of sharing a single document.

In experiments with desktop sharing in the Blue world, we find that a kilobyte per second is sufficient for applications which do not use of large color images. This rate is easily within the reach of all Apple network mediums, and is even achievable with 9600–baud modems.

Large color images pose a problem. A large color image (1280 h x 1024 v x 4 bytes) consumes 5 Megabytes of data, and requires 5 kiloseconds (85 minutes) to transmit at 1K baud. When faced with large color images, or complex graphics models, user will have to resort to strategies discussed in the Graceful Degradation section.

Document sharing ameliorates this problem, because the image need only be sent once, then it is available locally. If we send the document before notifying the receivers, we can hide the setup time from the receivers.

In some situations real-time sharing will be impractical. In these situations we can fall back to another mode of sharing: facsimile transmission. Users can collaborate by mailing each other screens, windows, and documents.

Leaving and rejoining a real-time collaboration

It takes some effort on part of the users and their machines to establish a real-time collaboration session. It would be nice to save the state of a session so that the users can continue long-term collaboration and resume real-time collaboration.

We have to decide what should happen if a machine or network involved in a collaboration should fail. In general it should be possible for the surviving collaborators to continue the real-time collaboration.

If the network is partitioned, it is possible that two independent groups of collaborators will result. Under the Cher document architecture, at most one group will have access to the model server. That's the group which can continue the real-time collaboration.

A user who is cut off from the model server could have the option of saving the cached model. This would create a new branch of the document.

Human Interface Issues

There are a number of human interface issues related to collaboration. The only way to resolve them is to go out and experiment with real users.

Long-term Collaboration Issues

Privacy, security, transparency

Privacy should be a guaranteed right of the ordinary Pink user. By default the user should have access to everything on their machine, and nobody else should have access to anything on their machine. It should take explicit actions on the part of the user to allow another user access.

But what about educational settings? I think we need a way of setting the global privacy policy for a machine.

Do we force people to log into their own machines? How often do they have to type their password? What happens when they forget their password? [13]

Encrypting communication between machines prevents unauthorized people from snooping, but it may significantly reduce the speed of communication. If it makes a big difference, how do we let the user choose wether to encrypt or not?

Access control lists

We need to allow selective access to documents, files, applications, and all other shareable resources. Whatever style of access control is decided upon, it should be consistent across as many different kinds of shareable resources as possible.

[13]The Apple Mail project is experimenting with the idea of "keys". A key provides authorized access to a service. Keys are icons which can be put on the desktop, stored in folders, given to other users, and thrown away. Just having a key is sufficient for using a service — you don't have to drag keys into locks.

People, group, & resources look & feel

People, groups, and other shareable resources should all be some sort of icons. People should look like little head&shoulder photographs of themselves. (What about Islamic countries, where they disapprove of representational art? Do they use ID Photos in their passports?) Groups should look like containers.

Shareable resources should be found and manipulated in the Finder.

Versions and branches of documents

When people work on a document over time, they create versions and branches. The major issues raised by versions are actually orthogonal to collaboration, and so fall outside the scope of this document. The HOOPS project is taking the lead on this issue. Collaboration raises the issue of access control.

Shared documents will be heavy users of the branching and merging capabilities of the version architecture.

See the HOOPS project, and the upcoming document version architecture for more details.

Real-time Collaboration Issues

Remote Mouse look & feel

Ideally, each user should see the pointers[14] controlled by all of the other users. These pointers should act just like the users own pointer. They should be ornamented with drop shadows and faces so that the user can tell the state of the remote mouse button and tell which pointer belongs to which user.

Since smooth pointer motion is necessary primarily as an aid to hand-eye coordination, it may be acceptable if the remote users' pointers moved more jerkily (say at 10 Hz) than the local user's pointer.

If it is not possible to support a large number of pointers, some alternative method of indicating remote users' pointer movements will be required. At the very least the pointer belonging to the user who is in control should be clearly visible to all collaborators.

Butler look & feel

Butlers mediate between users and the outside world. The word "butler" conjures up a middle-aged formal servant. This might not be the best user interface. It isn't entirely clear if an anthropomorphic agent is required at all. Prototyping and user testing is required here.

[14]Annette tells me that only programmers call these things "cursors".

Turn Taking

As far as the ordinary Pink application is concerned, there is only one thread of control. The real–time collaborators take turns being the "real" user.

There are many ways of taking turns. We must decide upon one which is low-overhead, but which isn't confusing. It may be necessary to offer a range of turn-taking policies, so that the users can choose the one which best fits their needs. Some possible policies are:

- Free-for-all
- Robert's Rules of Order
- Teacher & Students
- Baton Passing

Calling meetings

There must be a way of calling a meeting. This involves:

- selecting a group of people and other shareable resources
- notifying them that the meeting's occurring
- allowing them to respond
- gathering them into the collaboration

Once a meeting is in progress, it will be necessary for users and other resources to enter and leave the meeting.

Transmission of presence

During a real-time collaboration we want to the users to feel, as much as possible, as if they are all in the same place. This is why we provide multiple pointers and pictures of users. We should also provide for voice and video.

Voice is very important in real-time collaboration. The cheapest voice collaboration solution is provided by the telephone system. Once we have ISDN built into our computers, it will be even easier to use the telephone system to aid collaboration.

Voice digitizers, such as Farallon's MacRecorder may be good for annotations and voice-mail, but unless they can be used continuously in real-time they are not useful for real-time collaboration.

Video conferencing is harder to provide, because of processing and network bandwidth limitations. Never-the-less, we are already seeing experimental local-area-video-networks being installed to provide video–conferencing and to aid production of multi-media documents. Where these video networks are available it makes sense to use them for real-time collaboration.

Unsynchronized vs. synchronized views

Document sharing allows different users to view different parts of the same document. But once users are looking at different things, they may find it hard to return to a common ground. We will have to experiment with unsynchronized views to see if people can understand them.

Window Frames

When a document window is shared between several collaborators, we want to have some visual indication that it is shared. One suggestion is to put the user's faces in the title bar.

Inter-operability

"No man is an island, entire of itself; every
man is a piece of the continent"

—John Donne

Pink users will need to collaborate with people who do not have Pink machines. We must make this as easy as possible.

How well collaboration works depends upon what kind of machine the other person is using. From an inter-operability standpoint, there are four kinds of people a Pink user might want to collaborate with:

Kind of User	Description
Another Pink user.	This is the easy case. Everything should work. When Pink runs on several machine architectures we might have trouble shipping application object-code around.
A user of another collaboration-aware Apple product (e.g. Blue, A/UX, Newton)	This is harder because differences in toolbox support will rule out real-time document sharing. Real-time screen sharing should still be possible, although graphics may have to be translated into greatest-common-denominator objects like pixmaps or text.
A user of another brand of computer (e.g. VAX, Cray)	This is even harder than collaborating with non-Pink Apple products. Real-time collaboration might not be possible , but mail-based collaboration should work just fine.
A person who doesn't have any computer at all.	Although we can't send them digital information, we can still trade Fax, voice, and video messages.

What about different processor architectures?

Pink is designed to run on many different processor architectures. This means that Pink collaboration has to work between users who are using different processors.

Desktop sharing is easy between different processor architectures, because only the data is transmitted.

Document sharing is easy if versions of the same application are available for each architecture. Application borrowing is harder, because application binary files usually are targeted to a single processor architecture. Application borrowing would still be possible if applications were distributed in AppLex[15] style intermediate code.

Support for other styles of collaboration

Pink is designed primarily for the "knowledge worker". For that reason, the user interface supplied with Pink supports peer-to-peer collaboration. There are many other types of collaboration, and Pink should allow them as well. By keeping the lower-level classes as policy-free as possible, it should be possible for third-party developers to support other styles of collaboration.

Here is a short list of other styles of collaboration which will be common among Pink users. We might want to build support for some of these styles into the user interface. At the very least, we will try to leave the door open for third-party solutions.

Style of Collaboration	Description
Teacher / Student	The teacher can completely rearrange the student's machine. The teacher often needs to do the same thing to all the students' machines at once.
Parent / Child	Children love to play with direct manipulation interfaces. It is easy for a child to accidentally change setting or throw files away. We should build "Child-Proof Locks" into certain parts of the user interface.
Area Associate / Group	AAs need access to group member's machines for administrative and support tasks.
More than one User per Machine	It is very common for machines to be shared between two or more users. We should handle this situation more gracefully than we have in the past. Perhaps we will have to implement a login scheme.
Public Computers	Many libraries, schools, and small businesses provide people with access to personal computers. It should be possible to set up a Pink machine so that certain activities (such as deleting applications) are restricted.

[15]AppLex is Wayne Loofbourrow's experimental processor-independent object code. At the moment Applex programs run at about 1/4 to 1/2 the speed of native code.

Dependencies on other Projects

"You've got to have friends."

—Bette Midler

Collaboration works in Pink as a result of cooperation between a great many independently developed pieces of hardware and software. One of the reasons for writing this chapter is to bring all the interdependencies out into the open. With all of the pieces laid out, it is less likely that some vital capability will go unimplemented until the last moment. With all the issues enumerated, it is less likely that some vital issue will go unresolved.[16]

Related parts of Pink

Cher

The "C" in the acronym "Cher" stands for "Collaboration". It's a good thing, too! Without direct support from the document architecture, it would be impossible to implement real-time document-based collaboration. See the Cher chapter for more details.

Albert

Albert is the device-independent graphics rendering toolbox. Device independence assists collaboration by allowing people with different kinds of graphic displays to view the same images. In addition to device independence, Albert has to provide several other services to support real-time collaboration. Fortunately , these requirements are very similar to the features needed to support printers and graphic accelerators.

Albert's ability to add and remove GDevices at run-time makes it easy to implement desktop sharing. Desktop sharing is accomplished by inserting a new graph device which spans all of the shared screens. Albert sends rendering commands to all screens which intersect the area affected by a rendering command. This means that all rendering commands will be sent to the new graph device as well as the appropriate screens. The new graph device streams the commands over the network to the other machine. In this way, everything drawn on the shared screen is also drawn on the remote screen.

Font borrowing is accomplished in collusion with Albert's font manager.

[16]If you're an implementor of one of the projects mentioned here, and you don't think your project will supply the listed capabilities, please say so! If we're assuming too much, now is the time to tell us, so we can figure out what to do about it.

Multiple pointers greatly enhance the utility of real-time collaboration.[17] Each user controls one or more pointers. Even if only one user can edit the document at a time, the other users still need to be able to point and gesture. In addition, it should be possible to adorn pointers. Adornment adds information to the pointer image. The two major adornments we need are:

- Drop shadows to indicate remote mouse-button state.
- Little pictures which indicate pointer ownership.

It is worth noting, in passing, that screen sharing technology is very similar to screen recording and playback. This means that it should be very easy to support "video-tape" style help.

NetComm

The network is the main channel of communication between Pink machines. This means that the quality network haring services central to collaboration. Some specific things we need are:

- Network transparent clients & servers
- ADAS - Naming and Authentication Protocols
- Secure Password authentication
- Secure protocols
- Multicast protocols
- The Identity Browser

Events

Real time collaboration requires the ability to add and remove remote event sources without the involvement of the running applications.

Tracking

"Best Effort" remote tracking will require changes to the tracking system. The current (d10) tracking system will have to be overhauled to extend it in the following ways:

- Remote tracking
- Simultaneous tracking feedback in multiple views
- Tracking across view boundaries
- Tracking with more than one input device at the same time

Hopefully all of these features can be added in a clean, understandable fashion.

---

[17]Dan Venolia's Toy Box program is a compelling example of multiple pointers. It is a 3D collaborative environment. Each user has a 3D pointing device they use to create and manipulate blocks. All users are creating blocks in the same space, and all of the users can see each other's pointers and blocks. Two users can cooperate to change the size of a block by grabbing at opposite corners of the block and pulling.

Finder

The Finder provides all the user interface for starting collaborations. Specific features include support for:

- Finding other users
- Starting collaborations
- Rejoining collaborations
- Browsing Versions

Related Non-Pink efforts

"We must all hang together, or assuredly we shall all hang separately."

— Benjamin Franklin, at the signing of the Declaration of Independence

The following is an exhaustive list of related projects at Apple.[18] Hopefully Pink will incorporate the best appropriate features and ideas from all of these systems.

Project	Description
Apple Mail	Comprehensive store & forward mail service. Includes the ADAS authentication service.
AppleShare & FileShare	Server-based and distributed file servers.
Diet Coke	Publication & Subscription style of linking for System 7.0
Spider	Video–conferencing, screen sharing, and distributed device control.
Chorus	A suite of collaborative work applications.
BlipVerts	Agent based mail & document retrieval.
Newton	Collaborative component document architecture. Multi-cast network protocols.
Medley	A collaborative HyperMedia editing concept system.

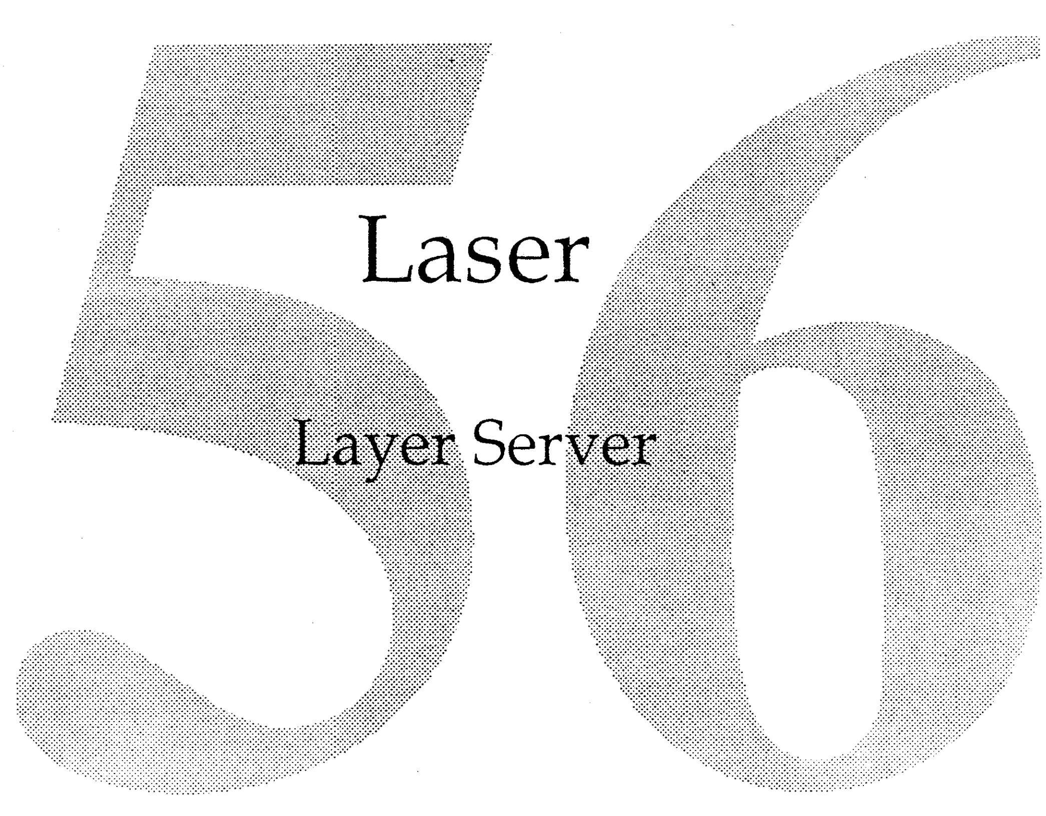
[18] If your project isn't on this list, it's because I haven't heard of you. Please drop me a line at PALEVICH1 or x4-4738.

Dependencies on hardware

So far we have discussed how collaborative software. Several hardware projects have the potential to greatly aid collaboration, too:

Project	Application
Sound I/O and Signal Processing	Voice–conferencing
	Audio Notes
Telephone interface	Voice transmission
	Answering machines
Fax interface	Fax-mail
Video I/O	Video–conferencing and picture transmission
Fast networks	Digital voice conferencing
	Digital video conferencing
	Sharing large documents
	Real-time collaboration with large documents

56



Laser

Layer Server

56

The Pink Layer Server

Larry Rosenstein
M/S 77-A x4-8123

Architecture

The function of the Layer Server is to divide the available screen area among all the applications that are running. You can think of it as a system-level window manager. (The View System would be the application-level window manager, since it manages the screen space within an application's layer.)

The direct clients of the Layer Server are generally adapters. (Applications, in turn, are clients of the adapters.) In the Pink world, the Toolbox acts as the "Pink Adapter." The View System in the Pink Toolbox includes a TLayer class that corresponds to a layer in the Layer Server. The Blue Adapter uses the Layer Server to implement MultiFinder layers.

Clients request screen space by creating a TSystemLayer object. To the client, a TSystemLayer represents an actual layer; when that object is deleted, the layer is also destroyed. The Layer Server provides two other classes (described below) that can be used to refer to an existing layer.

The Layer Server maintains a linear ordering for all the existing layers. Layers closer to the front obscure those behind it. The Layer Server computes a visible region for each layer, which is the layer's total extent minus the union of the extents of the layers above it.

The layer ordering is determined by three things. First, when a client creates a layer it can position it with respect to existing layers.

Second, the Layer Server implements different categories of layers. The layers of one category always appear in front of the layers of the following categories. This ensures that a menu layer, for example, always appears in front of a document layer, and that a desktop layer is always behind everything else.

Finally, the layer order is changed while the system is running. The most common way is by the user clicking on a layer. The Layer Server (in conjunction with the Event Server) enforces the policy that clicking on a layer brings it to the front of its category of layers. (Note that the layer does not necessarily come to the very front of the layer list; layers in other categories might still obscure it.)

It is also possible to programmatically bring a layer to the front of its category. (See the description below.)

Each layer contains a TSurrogateTask object, which is the task that "owns" the layer. The owner of a layer is the task that receives events for the layer. These events include mouse down/up, key down/up, and activate/deactivate events.

Key events are special because the keyboard (and similar devices) are non-positional. (In this paper, "keyboard" refers to all kinds of non-positional devices.) The target of keyboard events is determined by the state of the system, rather than the state of the input device. The Layer Server keeps track of the layer that gets these events.

The algorithm the Layer Server uses to determine the keyboard event layer is very simple, but seems to work for the typical cases. This is one area in which we need additional thought. (See **Open Issues** at the end of the paper.)

Only two kinds of layers can receive keyboard events: document layers and floating windoids. Document layers always receive keyboard events, while windoids may receive keyboard events. When the client creates a layer it specifies whether it can handle keyboard events. Internally, this flag is forced to TRUE for document layers, and FALSE for all other layers except windoids.

The Layer Server maintains the keyboard event layer using the following rules:

- If a layer is created or made visible, and it handles keyboard events, and it is in front of the current keyboard event layer, then it becomes the keyboard event layer.

- If the user clicks in a layer, and that layer handles keyboard events then it becomes the keyboard event layer.

- If the keyboard event layer is destroyed or hidden, then the frontmost document layer becomes the keyboard event layer. If such a layer doesn't exist, then the frontmost layer that accepts keyboard events becomes the keyboard event layer.

As mentioned earlier, TSystemLayer represents the actual layer, in the sense that the lifetime of the object matches the lifetime of the layer. The Layer Server also provides two ways to refer to existing layers. These classes provides alternate "names" for the same layers.

The first class is TSurrogateLayer. TSurrogateLayer is used to refer to a layer, and provides only a minimal set of capabilities. You can retrieve the owning task for a layer and flatten/expand the layer to a stream.

The second class is TLayerAlias. TLayerAlias provides complete access to a layer; the only differences between TLayerAlias and TSystemLayer are: (1) a TLayerAlias object can be flattened/expanded and can refer to different layers in its lifetime, and (2) TLayerAlias objects can be created and destroyed without affecting the actual layers on the screen.

In addition to retrieving the task owning a layer, a TLayerAlias (or TSystemLayer) object can do the following:

- change the extent of the layer,
- hide or show the layer,
- get the update and visible regions of the layer

The Layer Server arbitrates between layers by computing a visible region for each layer. Clients are responsible for getting the visible region and clipping all their drawing to at least that region. TLayerAlias (and TSystemLayer) maintain a seed to indicate when the visible region has changed. Clients can use this to take action only in the event of a change in the layer's visible region.

Since the Layer Server accepts requests from many clients, there must be some concurrency control to prevent any one client from using an out-of-date visible region. (The effect of this would be allow one application to trash the windows of another application.)

To implement concurrency control, the Layer Server creates a global drawing semaphore. When the Layer Server needs to update the visible region of any layer, it acquires the semaphore in exclusive mode.

56

Clients, using a method of TLayerAlias or TSystemLayer, acquire the semaphore in shared mode before retrieving the visible region, and therefore before drawing.

This protocol ensures that the Layer Server will not start recomputing visible regions until all its clients are through drawing, and that no client will begin drawing until all the visible regions are in a consistent state.

Client Interface

Global Types

```
typedef unsigned long LayerID;
const LayerID  kInvalidLayerID = 0;          // valid id's start at 1
```

LayerID is a unique ID assigned to each layer by the LayerServer. kInvalidLayerID is one value that is never assigned. (Unless of course the counter wraps around, which hopefully won't be a problem.)

```
typedef enum { kMenuBarLayer,
               kMenuLayer,
               kAlertLayer,
               kTornMenuLayer,
               kWindoidLayer,
               kDocumentLayer,
               kDesktopLayer } LayerKind;
```

LayerKind is used to describe the category of layer. The cases correspond (roughly) to: the menu bar, pull-down and popup menus, alerts, torn off menus, floating windoids, normal document windows, and the desktop.

```
typedef enum { kNewBackmostLayer, kNewFrontmostLayer } LayerPosition;
```

LayerPosition is used when creating a layer to indicate that it should appear at the front or back of its layer category.

TSystemLayer

TSystemLayer is the client's interface to the actual layers. The lifetime of the object matches the lifetime of the layer on the screen.

```
TSystemLayer(const TWorkRegion& itsExtent,
      TEventServer& anEventServer,
      const TSurrogateLayer& behindLayer,
```

```
                Boolean handlesNonPositionalEvents = TRUE,
                Boolean isVisible = TRUE);

        TSystemLayer(const TWorkRegion& itsExtent,
                TEventServer& anEventServer,
                LayerPosition itsPosition,
                LayerKind itsKind,
                Boolean handlesNonPositionalEvents = TRUE,
                Boolean isVisible = TRUE);
```

Create a TSystemLayer, which results in creating a physical layer.

Parameters:

itsExtent	The desired extent of the layer.
anEventServer	The "owner" of the layer; the task that receives the layer's events.
itsKind	The kind of the layer.
handlesNonPositionalEvents	Whether this layer handles non-positional (e.g., keyboard) events. This value is used only if itsKind is kWindoidLayer. If itsKind is kDocumentLayer it is forced to True internally; otherwise it is forced to False.
isVisible	Should the layer be created initially visible.

The different constructors allow you to specify the position of the layer with respect to existing layers. If you pass a TSurrogateLayer (behindLayer), then the new layer appears behind the layer. (If behindLayer is not of the same kind as the new layer's kind, then the new layer appears at the front of its category of layers.)

If you pass a LayerPosition, then the new layer will appear frontmost or backmost within its category of layer.

```
        ~TSystemLayer();
```

Destroy a TSystemLayer, which also destroys the physical layer.

```
    LayerID      GetLayerID() const;
    Boolean      IsValid() const;
```

Get the unique ID associate with the layer. This may be useful for creating another object (e.g., TSurrogateLayer), or if you want to pass the LayerID to another task. IsValid returns True iff the Layer ID is not kInvalidLayerID

```
    void         GetOwnerTask(TSurrogateTask& ownerTask);
```

Copy the task owning the layer into ownerTask.

```
    void         Hide();
    void         SetVisibility(Boolean makeVisible);
    void         Show();
```

These methods hide and show a layer. Hide and Show simply expand to calling SetVisibility.

```
void            SetExtent(const TWorkRegion&, Boolean addToUpdate);
```

Set the extent of the layer. addToUpdate specifies whether any new area belonging to the layer should be added to the update region of the layer. (Clients might pass False if they planned on immediately updating the new area.)

```
void            AcquireDrawingSemaphore() const;
Boolean         DrawingWaiters() const;
void            ReleaseDrawingSemaphore() const;
```

These methods deal with the drawing semaphore maintained by the Layer Server. Clients should call AcquireDrawingSemaphore before calling GetUpdate, GetVisRegion, or GetVisRegionSeed.

Since client must get the current visible region before drawing in the layer, they must bracket drawing with calls to AcquireDrawingSemaphore and ReleaseDrawingSemaphore.

```
void            GetUpdate(TWorkRegion& updateRegion);
```

Set updateRegion to the update region for the layer. This also clears the layer's update region.[1]

```
void            GetVisRegion(TWorkRegion& visibleRegion);
```

Set visibleRegion to the visible region for the layer. You must call AcquireDrawingSemaphore before before getting the visible region, to ensure that the Layer Server isn't in the process of recomputing a new set of visible regions.

```
Boolean         GetVisRegionSeed(TSeed& visRegionSeed);
```

Set visRegionSeed to the seed value for the layer's visible region. The seed acts as a version number for the visible region. You can use this call to determine if the visible region has changed or not since the last time you checked. You must call AcquireDrawingSemaphore before before getting the visible region, to ensure that the Layer Server isn't in the process of recomputing a new set of visible regions.

```
                operator TLayerAlias() const;
```

Create a TLayerAlias object from the TSystemLayer. A TLayerAlias provides a way to refer to a layer and fully manipulate it without affecting whether the layer is destroyed or not. A TLayerAlias object can be flattened and resurrected, while a TSystemLayer object cannot.[2]

---

1. If necessary, we can make clearing the update region an option to the call.
2. The reason is that you don't want to have two insances of TSystemLayer that refer to the same physical layer. Perhaps this is not a useful restrictions, in which case, we might be able to eliminate the TLayerALias class.

```
operator LayerID() const;
```

Return the LayerID of the layer. This is a convenience[3] operator so that you can pass a TSystemLayer to functions that take a LayerID.

```
operator TSurrogateLayer() const;
```

Create a TSurrogateLayer from the TSystemLayer.

```
long        Hash() const;
Boolean     IsEqual(const MCollectible* otherLayer) const;
```

Overriden methods from MCollectible..

TLayerAlias

TLayerAlias provides the same capabilities as TSystemLayer. The only difference is that the lifetime of the layer is not tied to the lifetime of the TLayerAlias object. An instance of TLayerAlias can be flattened and sent to another task, which can then manipulate the layer.

The interfaces below list only the method of TLayerAlias that aren't in TSystemLayer.

```
TLayerAlias(const TLayerAlias& otherLayerAlias);
TLayerAlias(const TSurrogateLayer& otherLayer);
TLayerAlias(LayerID itsLayerID = kInvalidLayerID);
```

Create a TLayerAlias from another kind of layer object, or from a Layer unique ID.

```
const TLayerAlias&    operator=(const TLayerAlias& otherLayerAlias);
```

Assign one instance of TLayerAlias to another.

```
void        SetLayerID(LayerID itsLayerID);
```

Get and set the unique ID associated with the object.

```
TStream&    operator>>=(TStream&) const;
TStream&    operator<<=(TStream&);
```

Flatten and unflatten operators.

---

3. In truth, it's a convenience for me. I can write methods that take a LayerID, and you can pass a TSystemLayer or other kind of layer object. I don't need to overload the methods to take both a LayerID and layer objects.

TSurrogateLayer

TSurrogateLayer provides a simple reference to a layer. You cannot manipulate the layer, but you can access some of its attributes.

The interfaces below list only the method ot TLayerAlias that aren't in TSystemLayer.

```
TSurrogateLayer(const TSurrogateLayer& otherLayer);
TSurrogateLayer(LayerID itsLayerID = kInvalidLayerID);
```

Create a TSurrogateLayer from another kind of layer object, or from a Layer unique ID.

```
LayerID     GetLayerID() const;
Boolean     IsValid() const;
```

Get the unique ID associate with the layer. This may be useful for creating another object (e.g., TSurrogateLayer), or if you want to pass the LayerID to another task. IsValid returns True iff the Layer ID is not kInvalidLayerID

```
void        GetOwnerTask(TSurrogateTask& ownerTask);
```

Copy the task owning the layer into ownerTask.

```
void        SetLayerID(LayerID itsLayerID);
```

Get and set the unique ID associated with the object.

```
TStream&    operator>>=(TStream&) const;
TStream&    operator<<=(TStream&);
```

Flatten and unflatten operators.

```
long        Hash() const;
Boolean     IsEqual(const MCollectible* otherLayer) const;
```

Inherited methods from MCollectible.

```
operator LayerID() const;
```

Return the LayerID of the layer. This is a convenience operator so that you can pass a TSurrogateLayer to functions that take a LayerID.

TSurrogateLayerServer

TSurrogateLayerServer provides access to several global functions of the Layer Server. These are functions that apply to the layers as a group, rather than an individual layer.

```
Boolean HandlePositionalEvent(const TGPoint& location,
                TSurrogateLayer& clickedLayer,
                Boolean& causedLayerSwitch,
                TSurrogateLayer& oldFrontLayer,
                Boolean& causedFocusSwitch);
```

Given a mouse location, this returns the layer that is under the mouse in clickedLayer. If there is no such layer, this call returns False.

If the layer was not the frontmost of its category, then the Layer Server will make it the frontmost and return True in causedLayerSwitch, and the previous frontmost layer in oldFrontLayer.

If the layer under the mouse location accepts non-positional events, then it will become the focus for such events and the Layer Server will return True in causedFocusSwitch.

```
Boolean LayerContaining(const TGPoint& location,
                TSurrogateLayer& theLayer);
```

Given a location, this returns the layer that is under that point in theLayer. If there is no such layer, this call returns False. Unlike HandlePositionalEvent, this does not change the layer ordering or the keyboard event focus.

```
Boolean NonPositionalEventLayer(TSurrogateLayer& theLayer);
```

Returns in theLayer the layer that handles non-positional events. This call returns False if there is no such layer.

Information Caching

The client interface to the Layer Server caches some information locally, in order to avoid sending requests to the Layer Server.

First, TSystemLayer, TLayerAlias, and TSurrogateLayer all maintain the owning task of the layer in the client object. If you create an object by supplying only the layer ID (not possible with TSystemLayer), then the client object records the fact that the task is unknown. The first time GetOwnerTask is called, the client object will send a request to the Layer Server to get the task. The cached task is also invalidated if you change the layer ID of the object.

Second, TSystemLayer and TLayerAlias cache the layer's visible region in the client object. Unlike the owner task, the visible region is modified by the Layer Server asynchronously from its client. When the

Layer Server changes a visible region, it updates a shared seed associated with the layer. The client objects examine the seed to determine if the cached region is still valid. If it isn't then the object sends a request to the Layer Server to get the latest region.

Clients of the Layer Server can also test the visible region seed. This is useful when a change in the visible regions requires some response from the client. For example, the View System recomputes each view's visible region when the layer's visible region changes.

Finally, TSurrogateLayerServer caches the layer that currently handles keyboard events. Again, the Layer Server updates a shared seed when this layer changes, and TSurrogateLayerServer only sends a request to the Layer Server when the cache is out-of-date.

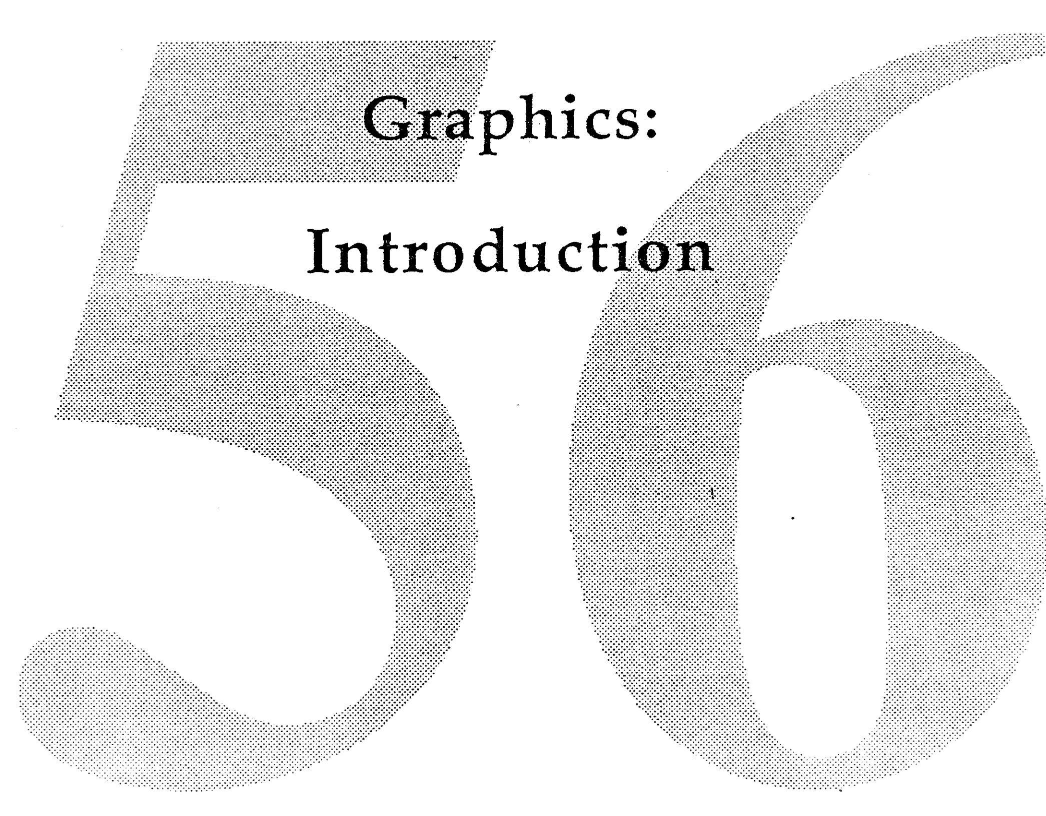
Internal Operation

<To be written. This will discuss the internal structure of the Layer Server and its classes.>

Open Issues

- Is the fixed set of layer categories sufficient?
- I don't particularly like having policy decisions built into the Layer Server. For example, the way in which the non-positional event layer is handled.
- If the Layer Server has to have the non-positional layer policy built-in, then is the algorithm it currently uses right?

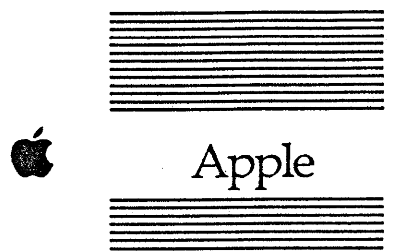
56



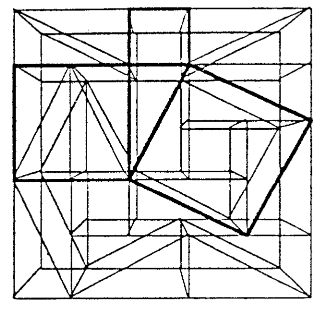
Graphics:

Introduction

56



Inside Albert



Architecture for Albert, the Pink Graphics System

Version:
Version 0.17, March 15, 1990, Pink Release 1.0d11, Section 2.3.2
Copyright © 1989, 1990 Apple Computer, Inc. All Rights Reserved.

Architecture and Documentation Authors:

Art Cabral	Editor, Introduction, Regions, GrafPorts,GrafDevices
Jerry Harris	Color (Bundles), Images
Shaun Ho	Shading (Bundles)
Mairé Howard	Transforms, Bundles
Don Marsh	Cursors (GrafDevices)
John Peterson	2D Geometry, 3D Geometry
Robert Seidl	2D Geometry, 3D Geometry
Roger Spreen	Graphics Collections (Introduction), Text
Victor Tso	3D Transforms

Albert Emeriti:
Gerard Schutten, Laurie Girand

Pink Architect:
David Goldsmith

Pink Architectural Contributors:
Bayles Holt, Ryoji Watanabe, Steve Milne, Larry Rosenstein, Mike Potel, Mark Vickers

Apple Architectural Contributors:
Bill Atkinson, Ernie Beernink, Steve Capps, Eric Chen, Jerome Coonen, Mark Cutter, John Dalton, Michael Kass, Lewis Knapp, Jon Magill, Gavin Miller, Steve Perlman, Steve Roskowski, Walter Smith, Galyn Susman, Ken Turkowski, Doug Turner, Kathryn Weisberg, Lance Williams, and Larry Yeager.

56

The Albert Graphics Architecture

Albert is the combined 2D and 3D graphics system developed in conjunction with the Pink system architecture. Albert accommodates the complexity of the current Macintosh hardware environment and provides additional extensibility and flexibility. By nature Albert is object-oriented. It uses a floating point number system to provide precision, range, and accuracy. It is device and resolution independent. It surpasses most of the combined functionality of Color QuickDraw, PostScript™, and Renderman™.

This section reviews the Albert architecture, beginning with Albert's numeric system and both the 2D and 3D coordinate systems, in particular showing how the two relate to one another. With that established, we list the geometries available in Albert, followed by the transformation matrix classes. We next provide a brief description of bundles, which are collections of graphical attributes, including paint, color, styles, and shading methods. The descriptions of geometries, transformations, and bundles set the stage for an introductory discussion of imaging with Albert, including a small example.

Numbers

Albert uses floating point numbers in all of its geometric data structures. Some objects, for example polygons, are constructed from an integral number of points, but coordinates are always expressed by the type **GCoordinate**, which is defined to be the floating point type **double**. *[For various reasons involving how ANSI C is specified, Albert currently defines* ***GCoordinate*** *to be the type* ***extended,*** *also known as* ***long double.*** *]*

Other numeric formats were considered. Fixed point (16.16) provides enough accuracy for most output devices but does not offer sufficient range for large documents, nor does it have enough precision to handle complex curves. Single (32 bit) float provides sufficient range but not sufficient accuracy. Double (64-bit) float provides sufficient range, precision, and accuracy, and so is the best compromise. *[Double fixed point (32.32) might be made to work, although the problems mentioned above would have to be examined carefully, especially with respect to curves and matrix mathematics.]*

2D Coordinate System : Y Goes Down

The coordinate system used by Color QuickDraw is also used by Albert. Positive coordinates along the X axis lie to the right of the origin and positive coordinates along the Y axis lie below the origin. The coordinate system is illustrated in Figure 1.

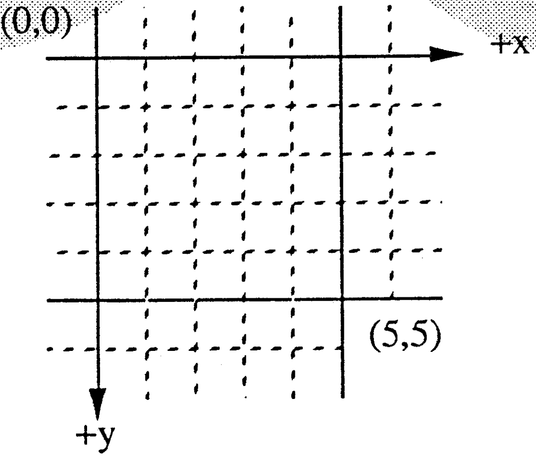


Figure 1. The 2D coordinate system in Albert

In Color QuickDraw a rectangle drawn from the values (1,1,4,3) would enclose exactly six pixels, as shown in Figure 2. Since all the original Macintosh displays had pixel sizes measuring 1/72 of an inch square, the area enclosed would be that of a rectangle measuring 3/72" in width and 2/72" in height. Albert assumes that a value of 1.0 will measure a distance of 1/72", but it does not assume that such a value will span exactly one pixel. In anticipation of new, higher resolution devices, Albert has been designed to be resolution independent. The Albert result for the same rectangle on a higher resolution screen (144 dpi) is shown in Figure 3. Note that although 144 dpi is convenient for this illustration, the resolution is not limited to multiples of 72.

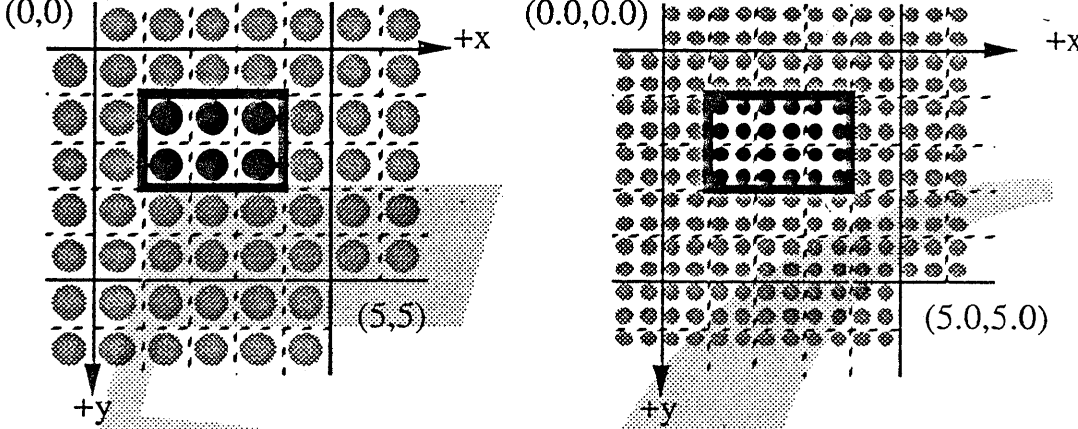


Figure 2. QuickDraw at 1/72" resolution   Figure 3. Albert at 1/144" resolution

The choice of a coordinate system matching that of Color QuickDraw has not been without controversy. An alternate that was strongly considered is that used by page description languages, for example PostScript™, where positive y values lie above the origin. The latter coordinate system makes certain types of imaging easier at the expense of others. In particular, the Pink View system assigns the origin (0,0) to the upper left corner of the view. This establishes a preference for the Color QuickDraw orientation so that most coordinates within a view will be positive in y, not negative. Since both systems are, ultimately, conventions, matching the Pink View system and Color QuickDraw was the most attractive option.

3D Coordinate System : Y Goes Up

The coordinate system used by most 3D systems is also used by Albert. Positive coordinates along the X axis lie to the right of the origin, positive coordinates along the Y axis lie above the origin, and positive coordinates along the Z axis lie towards the viewer. The coordinate system is illustrated in Figure 4.

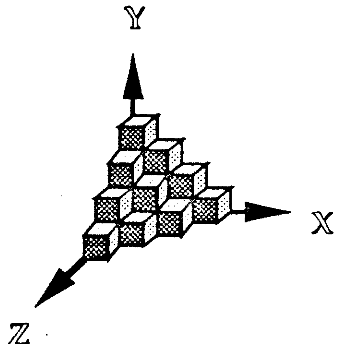


Figure 4. The Albert default 3D coordinate system.

The 3D coordinate system is at odds to the 2D coordinate system in two ways. First, positive values for Y lie above the origin, not below. Second, the origin is positioned by default in the center of windows, not in the upper left corner. Both differences reflect the preferences of most 3D developers. *[In practice, the inconsistency between the orientations of the 2D and 3D coordinate systems has not caused any major problems. By contrast, a prototype which placed the 2D origin at the bottom left did cause major headaches. Comments on this inconsistency are welcome, now. As time goes on, changes will become harder to make, so write early, and often.]*

The Integration of 2D and 3D

The best way to understand how Albert integrates 2D and 3D imaging involves an analogy to how a camera works. Consider Figure 5, below. We have created an interesting scene between to clipping planes, a near clipping plane and a far clipping plane. A camera works by gathering light rays and projecting them onto film behind the lens. Of course the image comes out upside down, but film is easy to turn over. For imaging 3D graphics the projection plane can be set in front of the camera to get an equivalent result. We then project all 3D geometries onto this plane, and that projection serves as the imaging result. In Figure 5 we have created such an image. It is also shown extracted from the camera model, after which we draw a 2D axis upon it. The final result is shown composited in a window. As you would expect, Albert's 3D camera can be manipulated in a number of different ways to control how the 2D and 3D systems interact. This will be discussed in the section on GrafBundles.

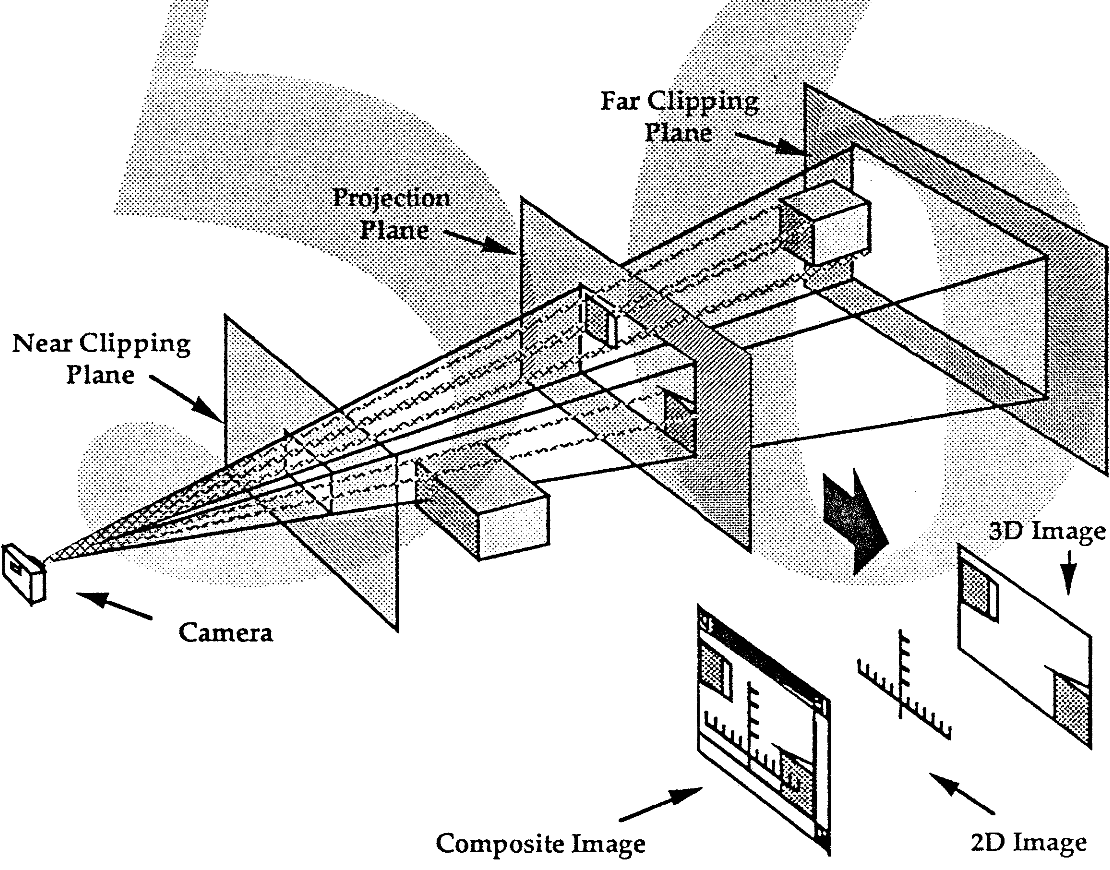


Figure 5. How Albert integrates 2D and 3D imaging

Points

Albert bases most of its geometric objects upon points. Every primitive can be constructed from points, and every primitive, other than the rectangle and the box, can be arbitrarily transformed to produce a valid geometric object of the same type.

Albert provides four types of points. The simplest is the 2D point, called **TGPoint**. It consists of two **GCoordinate** values, one for the x position and one for the y position. The 3D point, called **TGPoint3D**, has a value for the z position as well. 3D points are not derived from 2D points. To do so would imply that the two are directly related. They are not. It is quite true that a 2D point can be generated from a 3D point, but the process involves an explicit process of projection. Such projection requires additional information to be properly done. Specifically it requires a 3D matrix. It is not sufficient to simply assume that the z value can be set to some constant. This will, in general, give the wrong result.

The remaining two types of points add a rational component, called w. Albert provides a rational 2D point, called **TGRPoint**, and a rational 3D point, called **TGRPoint3D**. Rational points are most useful in controlling curvature for primitives such as nurbs and nurb surfaces. The utility of these points will be described in a separate document.

Points can be manipulated in a number of ways. They can be used mathematically very much as numbers can be used. For example, a pair of points can be added, subtracted, multiplied, or divided. For non-rational points these processes are simple extensions of the number operations to each component of the points.

Geometry

Like its architectural parent, Color QuickDraw, Albert lets you draw many different things on a Pink system screen. All of the familiar forms found in Color QuickDraw are available, as illustrated in Figure 6. *[Grayed out items are not yet implemented. All 2D items will be working within 2 months.]*

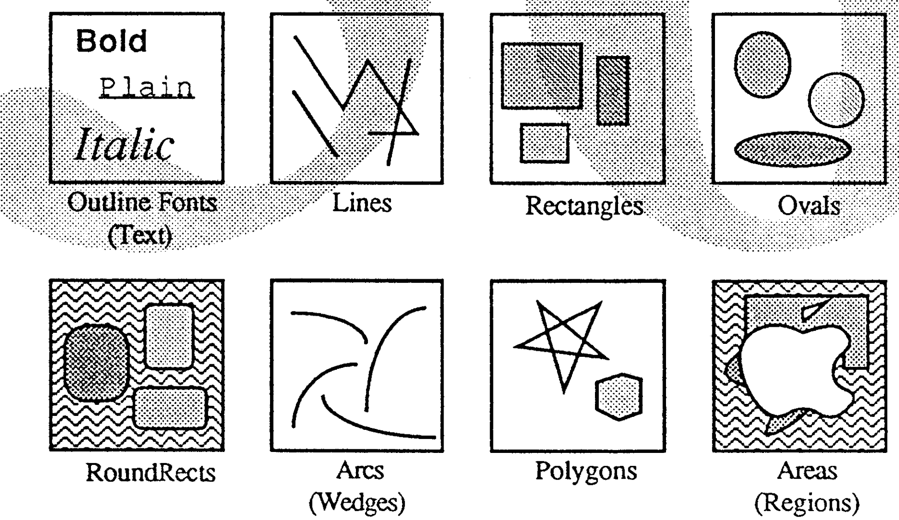


Figure 6. Geometric primitives inherited from Color QuickDraw

Albert extends Color QuickDraw's set of primitives by adding a curve primitive and a path primitive. Examples of these new primitives are shown in Figure 7.

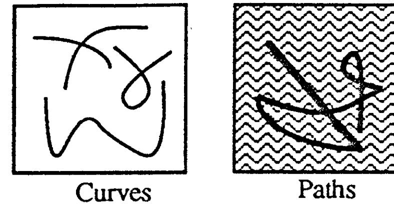


Figure 7. New geometric primitives provided by Albert

Albert actually provides three flavors of curves, as shown in Figure 8. Quadratic Beziér splines are familiar to users of MacDraw II® (Claris), while cubic Beziér splines are familiar to users of PostScript™ (Adobe). A nurb is a powerful specification for arbitrary curves of great complexity. The name derives from the acronym NURBS, which in turn stands for Non-Uniform Rational B-Spline.

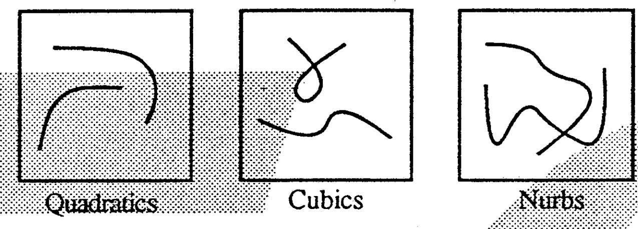


Figure 8. Curves provided by Albert

Albert extends the 2D world of Color QuickDraw to 3D. You can draw a variety of interesting 3D forms in Albert, as shown if Figure 9.

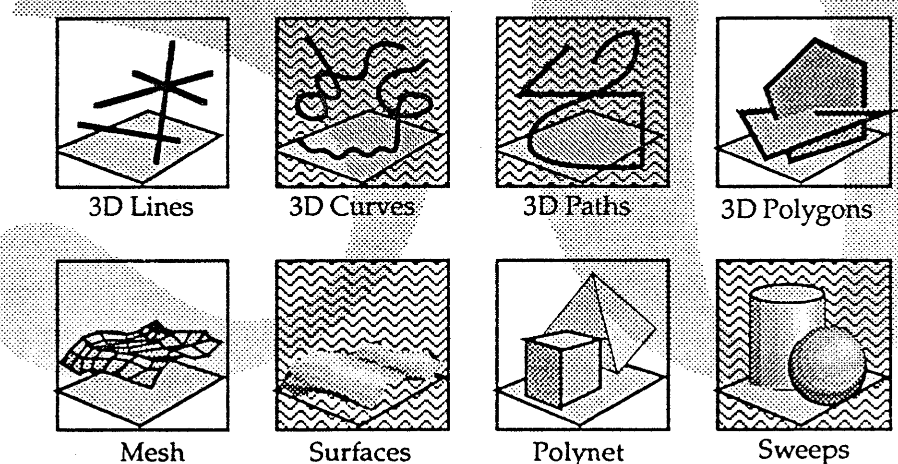


Figure 9. New 3D geometric primitives provided by Albert

Transformations

The most interesting way to manipulate a point is to transform it. 2D points are transformed using a 3x3 matrix, called **TMatrix**. Transformation involves a matrix multiplication, illustrated below in Figure 10 for a simple translation by 3 in x and 4 in y. The non-rational point, **TGPoint**, is assumed to have a third value of 1.0. The rational point, **TGRPoint**, uses its w value as the third value, and for some transformations will use the full 3x3 matrix, as illustrated gratuitously in Figure 11.

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} = \begin{bmatrix} (x+3) & (y+4) & 1 \end{bmatrix}$$

Figure 10. The transformation (translation) of a TGPoint.

$$\begin{bmatrix} x & y & w \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{bmatrix} = \begin{bmatrix} (2x) & (3y) & (4w) \end{bmatrix}$$

Figure 11. The transformation (scaling) of a TGRPoint.

3D points are transformed in an analogous manner using a 4x4 matrix, called **TMatrix3D**. In general, **TMatrix3D** will a used when you are imaging 3D objects and a **TMatrix** will be used when you are imaging 2D objects. Because 3D objects are generally rendered as a projection into a window on a 2D screen, most 3D imaging is subject to a 2D transformation as well.

Geometry and Transformations

When you create a geometric object in Albert, the process is very much like creating a stencil. The points define the limits of the geometry in a prescribed way. A line, for example, consists of a starting point and an ending point. It is called a **TGLine**. The naming convention **TGPrimitive** refers to a geometric primitive. Such classes generally include only geometric information, but not color or transformation information.

Consider a **TGLine** constructed from a point (1,1) to a point (5,5). There are two ways to create such a line. First, you can create the line directly, using a **TGPoint(1,1)** and a **TGPoint(5,5)**. You could also create a line from **TGPoint(0,0)** to **TGPoint(1,1)** and apply an appropriate transformation to translate the starting point to (1,1) and scale the line to the appropriate length. In either case the result is the same, assuming that numerical errors in the transformation calculations are insignificant. The first method, however, is generally easier to use as well as faster (since the transformation is avoided).

However, if you had started with a **TGPolygon** in the form of a star, and wished to place a series of stars around the perimeter of a circle, it would be tedious to precalculate the points for each of the star polygons. The easiest approach would be to create a single polygonal star and move it around the circle by manipulating a **TMatrix**. In this sense the polygonal star serves as a stencil, and the stencil is manipulated by using a matrix transformation.

Of the various 2D geometric primitives found in Albert, the rectangle, called **TGRect** behaves slightly differently. A rectangle is constructed from just two 2D points. The points are insufficient to specify any arbitrarily transformed rectangle. Instead the points define the top left and bottom right corners of a rectangle which lies orthogonal to the coordinate plane. If you wish to image a rotated rectangle you must use a transformation. As before, you can treat a rectangle as though it is a stencil. When a rotation is applied to the rectangle it behaves as though a stencil matching the orthogonal rectangle has been rotated to the new orientation before the rectangle is used (for example, drawn). This process is illustrated in Figure 12.

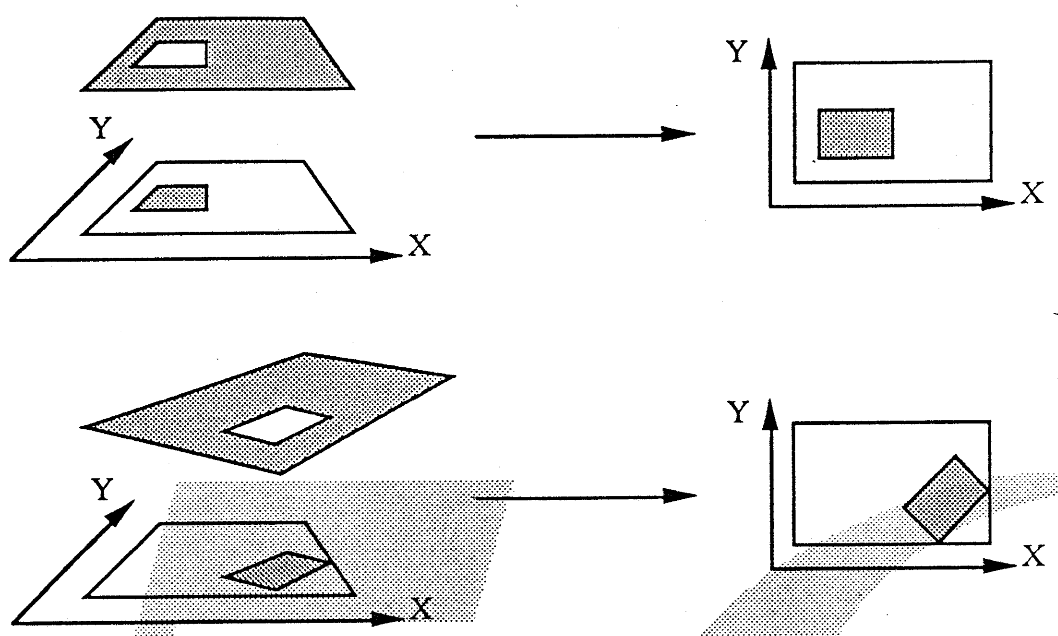


Figure 12. A stencil and a rotated stencil used to image a rectangle.

In the image above, a stencil for a rectangle is illustrated to lie above a drawing surface. The surface is also shown on the right when viewed from above. In the second image, the stencil is rotated before the rectangle is imaged, resulting in the rotated rectangle as shown. The advantage of this approach is that the rectangle data maintains its accuracy. It can be manipulated in a variety of ways, but its internal data is not changed. If two identical rotations are performed on the same rectangle, the imaging result will also be identical, even if a series of other operations are performed in the meantime.

The justification for having a rectangle of this sort lies in efficiency and utility. Many user interface elements require the imaging of untransformed (orthogonal) rectangles. For providing effective user interface this process needs to be as fast as possible. The simple structure represented by **TGRect** serves this purpose. In addition, many processes, such as hit-testing or user interface creation, involve establishing relationships among orthogonal rectangles. To facilitate such processes the structure of the rectangle needs to be as simple as possible.

But a rectangle is really just an accelerated polygon. To that end Albert provides a special way to construct a rectangular polygon which can, because it is a polygon, be manipulated in a general way. Note that some 3x3 transformations, for example perspective transformations, will cause a rectangle to lose its rectangularity.

The box primitive, called **TGBox3D**, has properties exactly analogous to the rectangle primitive.

Text, Glyphs, and GlyphRuns

Albert supports text at the most primitive level, *i.e.*, painting characters on the screen. With the ability to paint international text and symbols, the term "character" is inappropriate; what is normally described as a "character" may actually be made up of several distinctly separate pieces of area-enclosing geometry, called *glyphs*. For example, an accented character like 'é' could be made from two separate glyphs, one for the 'e' and one for the '´' accent.

More sophisticated textual functions, such as kerning, editing, composing characters from composite

glyphs, laying out words, lines, and paragraphs, etc., are performed at a higher level by clients of Albert. This is because line layout and editing functions are an especially broad, complex, interactive application of text, which is beyond the scope of Albert. Albert's responsibility is to be able to display text in any font, size, orientation, and style, as quickly and as easily as possible. To accomplish this, Albert text is divided into two areas, font management and glyph display.

Glyphs

The typical Albert user is primarily concerned about displaying glyphs. Fortunately, displaying glyphs is entirely consistent with the display of other Albert primitives. A glyph is simply another type of area-enclosing geometry. Glyphs are defined in *outline* form; that is, the glyphs are defined in terms of the curves that form the shape boundaries. This is in contrast to a *bitmap* form, in which the glyph is chopped into rectangles that represent the pixels of a raster image. The outline form can be scaled or rotated accurately, while the bitmap form is only useful at a specific size on a specific resolution graphics device.

For the purposes of placing the characters, each character has an origin for placing along a horizontal baseline, and each also has an origin for placing along a vertical path.

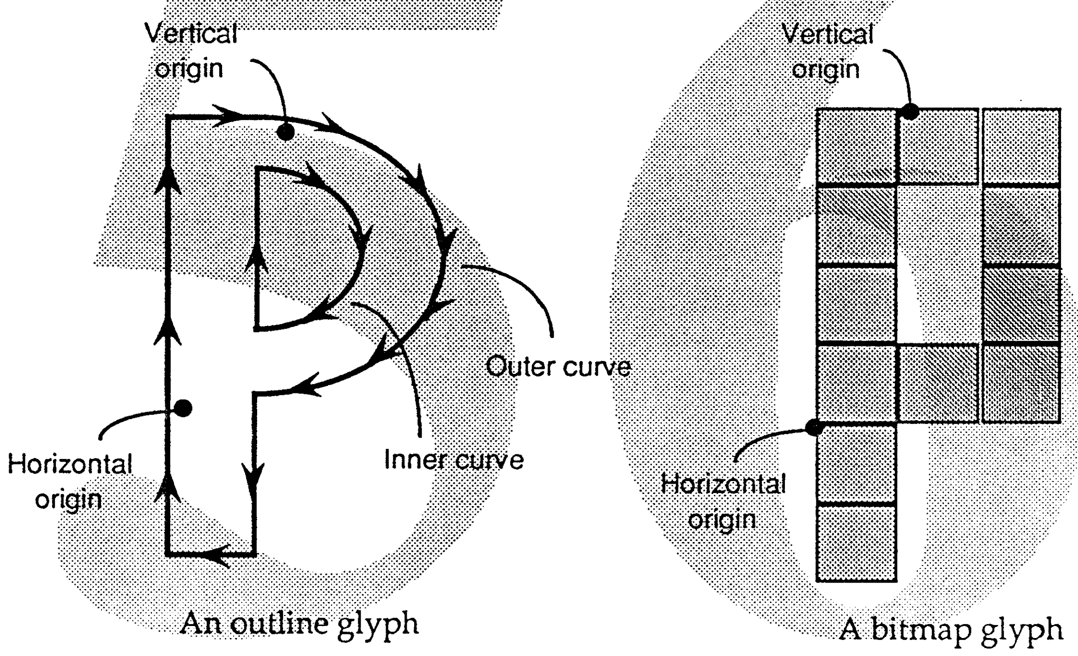


Figure 13. A comparison of an outline glyph and a bitmap glyph

Glyph Runs

Since text is most commonly displayed as a string of characters, Albert's basic text object is not a single glyph, but a collection of glyphs called a *glyph run*. The glyph run contains three types of information which completely determine the text's geometry. The first bit of information is a font descriptor, which includes information such as type face, point size, and style. The second piece is an array of glyph codes which index into the font. The third type of information provides for positioning in the form of an origin and an orientation for each character in the string. Orientation is recorded as a text path. The most common paths, such as horizontal or vertical, are recognized and handled as optimized special cases. However, Albert also allows you to place glyphs along arbitrary paths, where the arbitrary path is represented by the geometric primitive called TGPath.

Font Descriptors

The font descriptor (the TFont class) is the Albert client's connection to Pink's font mechanism; it contains the font name, base point size, and style information so that, given a glyph code, Albert can compute the exact glyph to be displayed.

Albert will support at least two font formats, namely bitmap fonts and TrueType (Bass) outline fonts. If a font is requested for which, on a particular raster device, a preferred bitmap form is available, Albert will substitute the bitmap form for the outline form. Albert will *not* attempt to modify a bitmap form into one of a different resolution or style. If the bitmap does not match the font descriptor precisely it will not be used.

A similar problem may arise in the event that data is imported from another system and a font is requested which is not available. Albert will not perform automatic font substitution, but will check for font substitution information stored in user preference information.

Bundles

Bundles are collections of graphical attributes. The first kind of bundle, which is called a **TGrafBundle**, is used to use collect attributes for one or more graphic objects. Consider the process of imaging a polygon in the form of a triangle. It might be filled with a pale lavender hue, or framed in a quiet cobalt blue, or both simultaneously. The frame might be a hairline or perhaps thick. It might be centered on the rectangular geometry, or inset, or outset. Perhaps the corners of the frame will be a bit rounded, or perhaps the frame will be dashed and dotted. All of these effects can be achieved by constructing an appropriate GrafBundle. If the standard styles don't accommodate your needs, you can extend styles in several ways to achieve special effects, such as the the last triangle in Figure 14.

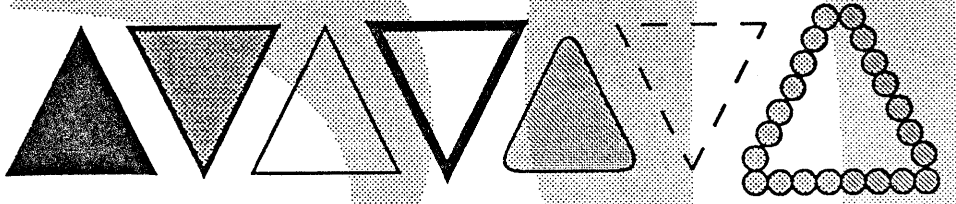


Figure 14. Triangles drawn in a variety of styles and paints.

Graphical attributes come in several flavors, each generally with its own class. The most commonly used attribute is paint, called a **TPaint**. Paints control how a graphic object is colored. There are several interesting kinds of paint, but certainly the most interesting is color, called a **TColor**. In the process of trying to standardize colors the graphics industries seem to have created a huge quantity of color spaces. Each of these can be derived from **TColor**, and many of them are predefined by Albert. These will be listed later, in the chapter describing bundles.

Some paints use more than one color. A pattern, for example, could include an arbitrary number of colors. Or a paint might be created which would use an image as a pattern. A ramp, from one color to another, is yet another interesting kind of paint. Transfer modes, including blending, also fall into the category of paint. But paints don't even have to involve color at all. A paint could be designed to write bits into a mask plane on a device which supports live video. The effect of using such a paint would be to cause live video to appear whereever the paint was used.

2D Attributes

Some attributes are specific to 2D primitives. For example, there are a few 2D primitives which enclose area: rectangles, ellipses, round rectangles, polygons, glyphs, and areas. Each of these can be

either filled, or framed, or both, and which is done as a property of the GrafBundle. Several properties affect how all 2D objects are rendered. Line styles, endcap styles, and join styles can affect most 2D primitives. A line style object, called a **TLineStyle**, can be used to control dashing, whether the pen is centered, inset, or outset, and it specifies a pen width. An endcap style object, called a **TEndCapStyle**, can be used to control the form of endcaps on open-ended geometries such as lines and curves. A join style object, called a **TJoinStyle**, can be used to control the form of joins on objects which have joins, such as rectangles, polygons, or polylines.

3D Attributes

With a sufficiently powerful 2D system it is possible to create images with many of the characteristics of 3D images. The problem is, it's hard, not very intuitive, and if anything simple detail changes you may have a lot of work to do. The goal of most 3D imaging is to create a somewhat photorealistic picture of one or more interesting objects. The real power of this approach comes from modeling. If models can be turned into accurate pictures, then the problem has been reduced to one of creating interesting models. In most respects that is an easier problem to solve.

The 3D imaging world thus concerns itself with a number of attributes that have no real parallels in the 2D world. A 3D object is given a color, but the color has no meaning until a light is applied to it, just as in the real world. The color of a 3D object is translated into an imaging result through the action of a shader object, calle a **TShader**. Two Shaders must be defined to properly render a 3D object, one for the inside and one for the outside. Other options which affect the rendering process include setting options for interpolating surfaces and for culling backface surfaces.

Scene Attributes

The TPortBundle defines global attributes for graphics rendering. The average user does not need to be concerned with the TPortBundle: it will default as well. For the expert client of Albert, the port bundle provides some of the more advanced rendering controls.

Graphic Objects

The preferred imaging layer in Albert involves the imaging graphic objects. Such objects are derived from a mixin class called **MGraphic**. This class provides a graphic object with a drawing method, several hit-testing methods, and a bundle in which graphic attributes may be stored. The **MGraphic** class is generally mixed into a geometry base class to produce a graphic object. For example, **TRect** is the combination of **MGraphic** and **TGRect**. Likewise, **TLine** is the combination of **MGraphic** and **TGLine**.

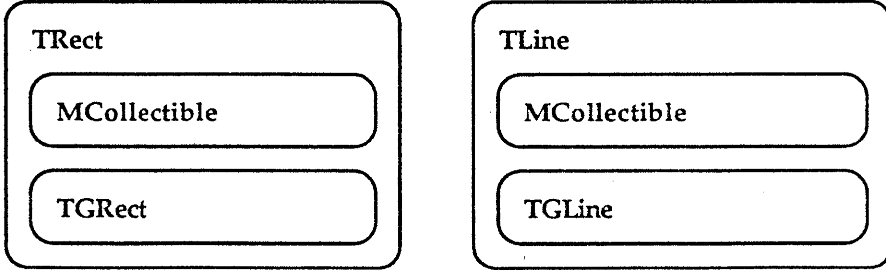


Figure 15. The class structures for **TRect** and **TLine**

Because graphic objects group geometry and graphic attributes, each one can be described as a small picture. But since each graphic object descends from **MCollectible**, they can themselves be placed in a collection, in this case a graphic collection called a **TGroup.** *[This section could not be finished prior to the publication deadline. Please refer to the next section, 2.3.3, which is included in this release and which describes graphic objects in greater detail.]*

Images

The Image object, called **TImage**, contains a graphics device which serves as an offscreen frame buffer. It can be manipulated as an **MGraphic.** You can also create a port based upon a **TImage** and draw to it just as you might draw to the desktop. *[This section could not be finished prior to the publication deadline. Section 2.3.8, which describes Albert's handling of images in greater detail, will be forthcoming.]*

GrafPorts, GrafDevices, and Regions

GrafPorts

The **TGrafPort** class provides synchronization with the toolbox along with some facilities for collecting and remembering graphical state.

GrafDevices

The **TGrafDevice** class is an abstract class which provides graphical imaging services. TGrafDevice rendering calls have no access to global state. The data for any imaging call, including geometry, color, style, transformation, and clipping data, is fully specified at the time of the call.

Albert can accommodate a wide range of graphics devices through this interface. Live video, for example, can be added to the desktop in a simple way, without rewriting an entire graphics device and without a major integration effort. Four plane animation hardware could also be implemented in a similar fashion. The exact details of implementing such devices will be described in a forthcoming section, 2.3.11, devoted to GrafDevices.

Regions

Regions provide a mechanism by which Albert and the View system can share a limited resource such as the desktop. Each ViewPort (GrafPort) built upon the desktop contains a Region which defines how much area it has reserved on the desktop GrafDevice.

The **TGrafRegion** class is also an abstract class. GrafRegions can be added, subtracted, intersected, or exclusively complemented (exclusive or'ed, but how the heck do you say that in English?). GrafRegions are created by GrafDevices, because only GrafDevices know which sort of GrafRegion will be appropriate. For example, the **TFrameBuffer** GrafDevice creates a GrafRegion of type **TRegionFrameBuffer.**

Architectural Principles

In the process of creating Albert, we spoke with a great many graphics experts at Apple, most of whom are acknowledged on our cover page. From these discussions we developed a series of Albert Principles. These are not offered as being original to Albert, but rather are included here because they are fundamental to the Albert architecture.

The Principle of Simplification

To quote Alan Kay, "simple things should be simple. Complex things should be possible." The goal is to have a simple process, such as drawing a line, to be a trivial process. You don't have to initialize anything, or start up the graphics system. Just get a GrafPort and draw a line.

YourApplication::Main()

```
TGrafPort* Port = new TViewPort(YourView);
TLine Line ( TGPoint(10,10), TGPoint(20,20) );

Line.Draw(Port);
```

The Principle of Orientation

Which is to say, object-oriented. Since Albert's reason for existing is to be the graphics system for the Pink system architecture, and since that architecture is thoroughly object-oriented, it behooves Albert to follow in kind. That doesn't mean just using objects and classes, but using them in a manner consistent with the rest of the Pink architecture.

The Principle of Inversion

Using an object-oriented approach, however, does not mean that Albert follows the model proposed in nearly every book on object-oriented programming yet written. Graphics examples seem to be the number one choice for most authors. The authors treat geometric objects as simple objects which know, among other things, how to draw themselves. Consider the (non-Albert) example shown in Figure 16.

YourApplication::Main()

```
TRectangle Rectangle ( 40, 40 );

Rectangle.Move ( 30, 30 );
Rectangle.Draw ();
```

Figure 16. A common object-oriented programming example.

While this approach makes for a nice example, it is clear that the authors haven't actually tried to do much with it. They certainly didn't try to accommodate a hardware platform as robust as the current Macintosh. The problem can be stated simply: how does the rectangle from Figure 16 understand how to

draw itself in an environment far more complex than the one for which it was originally designed? The authors have presumed a static environment, where the process of drawing a rectangle never varies. This can be easily thwarted in the Macintosh environment by simply adding another monitor to the desktop. Of course, the rectangle object could be designed to understand multiple monitors. But then what about graphics accelerators, or video, or the neat new graphics hardware announced tomorrow?

It becomes clear that the most interesting object in the Macintosh hardware model is the graphics device, not the geometries. Regardless of what type of hardware it is, a graphics device isn't interesting unless it can draw geometries such as the rectangle. Exactly how that occurs is important to the graphics device, but not to the rectangle. Therefore the foundation of imaging in Albert looks, although not exactly, like the example code shown in Figure 17.

YourApplication::Main()

```
TRectangle Rectangle ( 40, 40 );

Rectangle.Move ( 30, 30 );
GraphicsDevice.Draw ( Rectangle );
```

Figure 17. The foundation of imaging in Albert

Having solved the problem of imaging on a variety of graphics devices, Albert can still provide an interface similar to the one in in Figure 16 by inverting the calling sequence described in Figure 17. This inversion is applied between the graphic objects and the geometries, which are imaged by a graphics device through the services of a middle layer, the graphics port. Graphics objects have a draw method which requires a parameter that tells *where* the drawing should be done. The parameter is in the form of a graphics port. The graphics port is built upon and has access to a graphics device. The final result, which will be described later in greater detail, is shown in Figure 18.

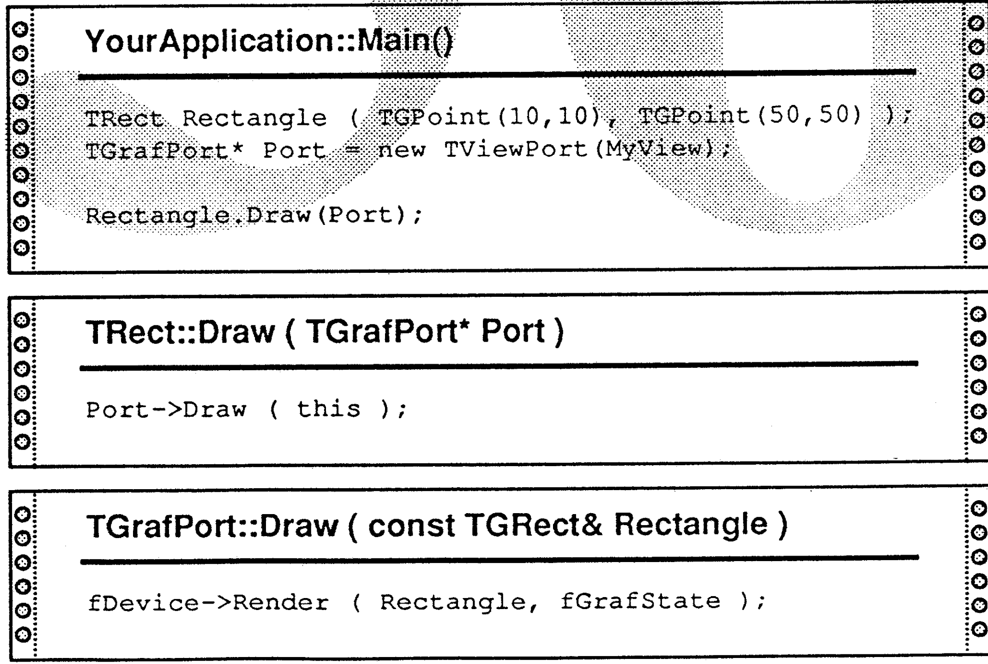


Figure 18. The three imaging layers of Albert

In Figure 18, the imaging call denoted by "Rectangle.Draw(Port)" is the preferred imaging layer. As will be described later, the **TRect** object actually includes more than just geometry, so the model portrayed above is somewhat simplified from the real model. In fact, the geometry portion of **TRect** is a separate base class called **TGRect**. However, the figure serves to illustrate the inversion from a TRect which knows how to draw itself (given a **TGrafPort** "where" parameter) to a **TGrafPort** which knows how to draw a **TGRect**. The **TGrafPort** adds certain pieces of information to the imaging process (for example, window positioning) and passes the call to the graphics device upon which it was built, a field **fDevice** of type **TGrafDevice**.

The Principle of Justification

All of which leads to the principle of justification. In the examples above an extra imaging layer snuck into the discussion, namely that of the **TGrafPort**. The principle of justification demands that each layer in the hierarchy provide value.

The top imaging layer, that dealing with graphic objects, provides polymorphism and extensibility. It is the only one you should be using for most applications. Graphic objects of this layer all descend from a class called **MGraphic**. The properties of an **MGraphic** include drawing, hit-testing, and manipulating attributes such as color and style. The **MGraphic** itself does not contain geometry, though. Each graphic object is derived from a simpler base class. For example, TRect inherits behavior from both MGraphic as well as TGRect.

There are many geometric objects of the form **TGPrimitive**. Examples are **TGLine**, **TGCurve**, and **TGPolyNet3D**. The corresponding graphic objects would be **TLine**, **TCurve**, and **TPolyNet3D**. However, more than one graphic object can be built upon a particular **TGPrimitive**. For example, a diamond-shaped polygon called **TDiamond** could be built upon the class **TGPolygon**. In fact, many common shapes, such as **TSphere**, **TCone**, and **TCylinder**, are built upon the single geometric object called **TGSweep**.

The grafport layer lies in the middle. It images geometric objects and provides synchronization and locality. This is done by collecting certain kinds of graphics state, for example window positioning and clipping, into the object described earlier called the **TGrafPort**. The grafport is the primary vehicle for interaction between the Pink View system and Albert. The View system derives a special type of grafport, called a **TViewPort**, which adds synchronization behavior so the windowing system can function correctly in a multi-tasking environment.

The grafdevice layer lies at the bottom of the hierarchy. Your application should not make calls directly to a grafdevice. But then, you should not be making calls directly to a grafport, either. The grafdevice layer provides Albert's contract with both internal and external developers of graphics devices. The graphics device developer need implement a fixed number of imaging calls in order to accommodate the full range of Albert's graphics capabilities.

The Principle of Localization

Which leads us to the Principle of Localization. The key to imaging with Albert is that at the time of an imaging call to a grafdevice, such as the `Render(Rectangle, fGrafState)` call in Figure 18, all information relating to the imaging process has been localized in the parameters to the call, the geometry in `Rectangle` and everything else (color, style, shading, transformation, and clipping) in `fGrafState`.

Perhaps one of the most important lessons we learned from QuickDraw (actually, from our esteemed colleagues who made QuickDraw print) is that it is important to gather all of the information used in the imaging process into one place at the time of the imaging call. In QuickDraw the classic problem is one of developers making modifications to the data structures of a grafport without using access methods. The change in state thus does not get recorded in a manner that the graphics system can accommodate it.

In Albert , the problem of hidden change is defeated in two ways. First, there is no equivalent to QuickDraw's SetPort(grafPort) call. All imaging references a specific grafport. In fact, Albert maintains no global state whatsoever. There is still room for developers to get into trouble. They could, for example, discover ways to change fields in Albert's various graphical attribute objects without using access methods. Albert solves this problem by ignoring such changes.

Principle of Abstraction

Closely related to the Principle of Localization is the Principle of Abstraction, which, stated another way, means "you don't really need to know," or "pay no attention to that man behind the curtain." Perhaps the best example of this is the region class, called **TGrafRegion**. QuickDraw provided regions, and quite useful beasts they were. QuickDraw also patented regions, making certain kinds of extensions a notoriously tricky thing to do for the Macintosh platform.

Albert certainly makes use of regions, and we have our own implementation based substantially on the QuickDraw region patent. However, we make only a small number of assumptions about regions in the course of using them. There are certain operations we need to perform to get the correct windowing and imaging results. For example, regions may need to be added, or subtracted, or intersected. Albert doesn't need to understand how a region is implemented in order to work with them. We only have to guarantee that the implementation supports the operations we intend to use. That can be accomplished by correctly designing the region interface.

Of course, to accommodate our own graphics devices we'll need a particular instantiation of the region class. For that matter, the **TGrafDevice** class is also an abstract class which supports imaging calls. While we also need to instantiate several grafdevices to support Apple hardware platforms, the grafdevice interface need not understand that implementation. The primary instantiations we provide for each of these important classes are called **TRegionFrameBuffer** and **TFrameBuffer**, respectively. The two classes have intimate knowledge of one another but do not talk about it in public.

The advantage of this approach is that external developers can use it. If for some reason they have created a piece of hardware which is not a frame buffer, and therefore upon which running **TFrameBuffer** code would be inappropriate, they can create their own grafdevice which satisfies the **TGrafDevice** interface, say **TWhizzyGrafDevice**, and all will be well with Albert. Of course, it is likely then that **TRegionFrameBuffer** will also be inappropriate, so they'll probably need a **TRegionWhizzyGrafDevice** as well.

Principle of Declaration

So, in principle, it's better not to know what's going on behind the curtain. Which creates a bit of a problem, because traditionally Macintosh developers like to peek in on what's going on back there. The trouble is, it won't always be possible to look. If your application is trying to be clever and check up on all grafdevices in the world and do the best thing for each one, you are guaranteed to encounter a situation you haven't anticipated, meaning **TWhizzyGrafDevice** which was announced a couple of days after your application went final. At best, your application won't draw properly. At worst, it will crash or fail in some way.

The solution to this dilemma is to rearrange your thinking on what you're trying to do. In general, you have a particular goal in mind, and the motivation for peeking behind the curtain comes from wanting to do the right thing in each particular case. The problem can be turned around, though. Suppose you state, in some predefined way, what your goal is. For example, you wish to draw a rectangle which has lines inset from the actual perimeter of the geometry and which all appear to be the same thickness. The temptation is to check up on the device resolution, anticipate out how rounding will occur, and specify the appropriate line thickness for each part of the rectangle in order to get the appropriate result. Unfortunately, it turns out that **TWhizzyGrafDevice** is a graphics accelerator of unknown composition, and it doesn't make its resolution available, perhaps because the concept of resolution has no meaning to the device.

What you really want to be able to do is to declare your goal for a particular imaging call, and allow the device to achieve your result by whatever means it has available. Albert supports this methodology by providing a flexible and extensible set of graphical attributes. However, determination of the initial set of available declarations is also one of the most challenging problems we face.

Principle of Optimization

In the early stages of the QD2 project (now called Skia) Cary Clark and Bruce Leak created a map of the decision tree in Color QuickDraw. Along the path of drawing a simple object such as a rectangle lay a great many decisions. Many of these decisions were found to be unnecessary; they had already been decided earlier in the path of drawing, but due to convergence of paths, the information learned along the way had been lost or discarded.

The Principle of Optimization can be stated as "make a decision exactly once." A number of approaches are available to satisfy this goal. For Albert we have designed the path of drawing a simple object to get to the rendering process as quickly as possible.

Take for example the problem found in Color QuickDraw with respect to variable bit depth monitors. After discovering that an imaging call has been made, Color QuickDraw must still examine the target grafDevice and select the set of imaging routines appropriate to the bit depth. In Albert, the GrafDevice is an object which already knows which imaging routines to use by the time the imaging call reaches its code. The decision is made once, at the time the desktop is initialized, by creating a GrafDevice of a type tailored to the particular monitor upon which is resides.

Or, for example, consider the case of multiple monitors. A GrafCluster GrafDevice is used to manage two or more GrafDevices. When a window is built upon such a cluster, its GrafPort contains a GrafRegion which reserves space on each device as necessary. If, as is likely to be the case, the window occupies space on only one device, the GrafRegion will understand and remember this information. When an imaging call occurs, the GrafCluster first makes a trivial check to see if there is any possibility of drawing to a device by asking the GrafRegion if any space has been reserved. If not, the imaging call is never made to that device.

One trick we use to make decisions only once involves arranging the decision tree such that decisions are made at the proper time. In general, the proper time is determined by minimizing calculations for the most frequent imaging calls. For the Macintosh system these calls involve simple geometry, including text, rectangles, lines, and copied images (although not necessarily in that order). The most common graphical attribute is color, in particular solid colors, especially black and white. The most common transformation is a simple translation, such as the screen offset of a window. The most common clipping is unclipped. The decision tree is established with these frequencies in mind. The most frequent cases are checked first.

Consider our implementation of the frame buffer GrafDevice. If a rectangle is being imaged, the GrafRegion of the GrafPort in use is asked whether the rectangle is clipped. If not, the flow of control passes to a routine which optimizes the unclippped case. No further consideration is made for clipping along that path. If the rectangle is being clipped, a different path is taken which reads the GrafRegion structure to produce the correctly clipped result. The clipping descision is made exactly once, and is made by the entity which understands clipping, namely the GrafRegion.

Principle of Acceleration

Another way to achieve optimal result is by using accelerated forms. For example, a rectangle could be expressed as a polygon, and could be imaged correctly in all cases. Rectangles, however, are imaged frequently, and can be handled in simpler, faster ways than general polygons will allow. Recognizing this, we offer rectangles as a separate primitive in Albert.

We handle curves in a slightly different way. Albert offers three flavors of curves, namely quadratic Bézier splines, cubic Bézier splines, and nurbs (from NURBS, or Non-Uniform Rational B-Splines). However, curves appear as a single object type with three different sorts of constructors, one for each of the types listed above. During the imaging process, however, our frame buffer GrafDevice recognizes each of the different forms and images each one in a slightly different way, resulting in optimal performance for each type. In the case of curves, there is no advantage for having a separate class for each type, but a great deal of advantage (at the frame buffer level) for imaging each type in a slightly different way.

A third example involves how Albert handles 2D transformation matrices. In common use, most transformations consist of simple translation, or, for high resolution devices, a combination of translation and scaling. The TMatrix class is designed to optimize for all important cases. Consider the operation of transforming a point. For the identity matrix, the call to transform a point immediately returns the point unchanged. A translation matrix simply adds an offset to each component. A scaling matrix multiplies each component by the appropriate scaling factor and adds an offset to each component. Only complex (non-affine) matrices need perform the full 3 x 3 matrix multiplication, and such matrices are used infrequently.

Principle of Procrastination

Another principle which provides a degree of optimization is lazy evaluation, or "put off until tomorrow whatever might get cancelled anyway." In Albert, we sometimes record logical results as opposed to calculated results. For example, in the implementation of Albert's generic 2D shape primitive, called **TGArea**, you might arrange for a geometric object to consist of the intersection of two complex closed curves. Rather than performing the intersection, we record both curves and the logical operation of intersection in the Area's data structure. If your next operation happens to overlay both curves with a rectangular area, we can simply remove both curves as well as their logical operation, having never done the complicated intersection calculation.

Principle of Perfection

We can do better than that, though. If in fact we *never* perform the intersection we can produce a more accurate result. Take for example two interesting shapes, a complex curve resembling a flower and one composed of concentric rings.

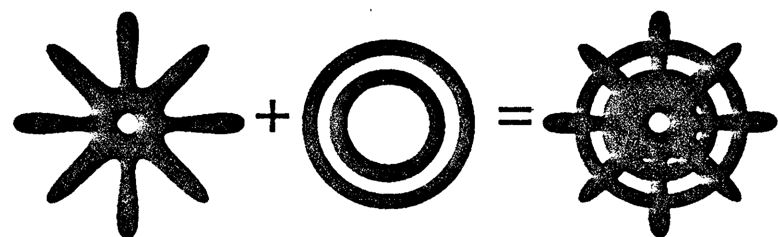


Figure 19. Addition of two TGArea objects to create a more complex form.

The addition of these two objects creates an interesting geometric form, but calculating the precise geometric form can be expensive and tricky. However, rendering each form independently is a well characterized process. If we delay the intersection process until rendering time, we can deal with a much simpler problem. In the case of a frame buffer we can deal with discrete pixels. In the example above, a pixel will fall in one of the two areas, and be drawn, or not. We never have to calculate the precise intersection of the two complex areas in order to render the addition result.

At the same time, we dodge another bullet. Even with high-precision floating point it is possible to introduce numerical errors via the intersection calculations which would lead to a different result for each object being imaged separately, and overlaid, versus imaging the added result from the Area. Delaying the process of intersection allows us to produce the best rendering of the "perfect" result.

Principle of Substitution

Albert does not perform font substitution. If a font is missing, such as when imported data specifies a font the user does not have, the system must resolve the missing resource, possibly through consultation with the user. Such consultation may result in a substitute font being selected, but the result of that is to direct Albert towards a different resource as opposed to trying to stretch or shrink a similar font into the right size.

Principle of Precision

High-precision floating point has its place, though. Color QuickDraw's 16 bit integer range was useful for a great many wonderful applications, but it was limited in three important cases. First, it failed to provide support for most modeling applications, which require floating point precision. Second, it was quite easy to create documents large enough to span the entire coordinate plane. After all, a span of 65536 pixels is only about 83 letter size pages at 72 dpi. At typical printer resolutions the coordinate plane is exhausted after only 21 pages.

Two 32-bit solutions were considered, namely fixed point integer (16.16) and single precision floating point. The first has improved resolution with sub-pixel accuracy but has no more range than Color QuickDraw. The second offers much improved range, but precision varies significantly depending upon which part of the coordinate plane is used.

After much pontification of this problem our favorite numerics expert recommended 64-bit double precision. Although a tad bit on the heavy side at 8 bytes, **double** provides the best balance among precision, range, and size. It also allows us to provide rendering of higher order primitives such as nurbs and nurb surfaces and to create well-behaved matrix classes.

Principle of Indiscretion

In this case, indiscretion means non-discrete. The Principle of Indiscretion means simply "use higher

order descriptions of geometry whenever possible," A quadratic Beziér spline, for example, can be described in a number of ways. A series of short, connected line segments which follow the curve can give excellent results on some graphics devices. However, if the resolution of the graphics device is greatly increased, it will at some point exceed the resolution used to select the line segments themselves. At such a point the intersections of the line segments will become visible, as illustrated in Figure 20

A better approach is to represent the quadratic Beziér spline as three control points. Even if the rendering technique involves the creation of short line segments, if the process of creating the line segments can be delayed until the resolution of the device is known, we can be assured that the resolution used to create the line segments will be sufficient to produce the correct imaging result.

This approach is even more important to 3D imaging. The analogous problem involves the representation of 3D surfaces. The less preferable solution involves using a series of small, connected 3D polygons to describe them. At high resolutions the edges of the polygons appear. Consequently, a number of polygon shading techniques have been devised in order to smooth away the edges and make the surfaces appear photorealistic.

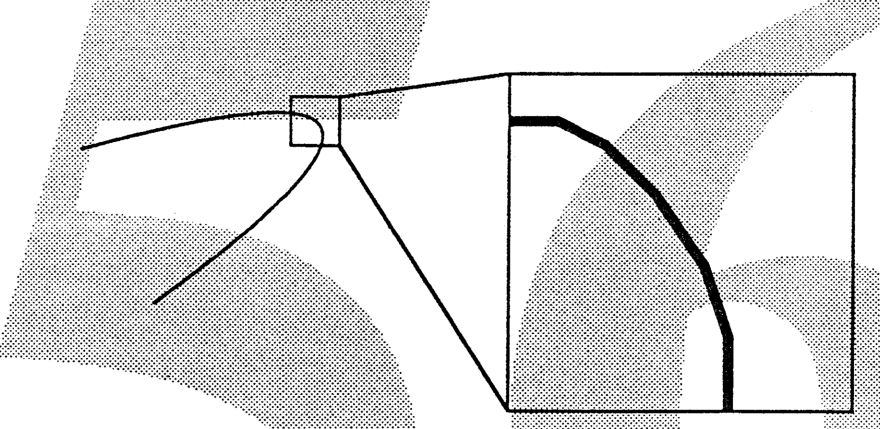


Figure 20. Using line segments to represent a quadratic Beziér curve

However, consider the problem of using Albert's imaging capabilities to drive lathes or other types of sculpting machinery. The smoothing techniques which work well for the screen are no longer available. Since the data is represented as polygons, only a faceted object can be produced. The alternative of using control points to specify surfaces leaves the problem of rendering to the graphics device. The polygon solution is still available, if appropriate, but other, higher quality solutions are not prohibited. The sculpting machinery might use the control points to form an analog function for moving across the surface to produce the desired result.

Principle of Integration

Albert was designed to integrate the 3D system with the 2D system. It isn't always possible to maintain consistency between the two worlds because imaging in each is fundamentally unique. For example, the most popular choices for orientation of coordinate systems in the two worlds do not match. Points and transformations, while somewhat related, are used in different ways in 3D as compared to 2D. And, perhaps most importantly, the goals for each kind of imaging are quite dissimilar.

But there are a great many things we *can* do to integrate the two environments. First and foremost, you don't have to make any special calls to start up the 3D system. It's there and ready to go as soon as you get a GrafPort for drawing, just like the 2D system. Because each system uses the same numerics, conversions between 2D and 3D data structures can be done in a straightforward manner. Objects, such as colors, are shared between the two worlds. Naming conventions, and the kinds of calls made to image each type of object, are consistent.

Principle of Organization

The Principle of Organization states that "display lists are a bad idea. There shall be no display lists in Albert." Well, maybe one or two. The basic problem with display lists, especially those at the device level, is that writers of most interesting applications will want to organize graphical data according to the particular needs of the problem being addressed. If the system maintains its own display list it pretty much guarantees that two copies of the data will be maintained at the same time. Silicon Graphics went down this road for awhile, then turned around and came back when it proved untenable.

However, the preferred imaging layer of Albert sure looks a lot like a display list. It is hierarchical, has groups and instances, and can be imaged with a single call. But here's a secret, just between you and Albert. Don't tell anyone else. The top layer of Albert is optional. The top layer of Albert is called the **MGraphic** layer, because graphical objects each descend from that (mixin) class. It is built completely upon the GrafPort layer. Since the GrafPort layer is also visible to developers, it could be used to build an alternate display list, one which maintains the data in a different form or which is used in a different manner. But we consider this to be the exceptional case. Many application authors will want to use some graphics, but their applications will not be graphics intensive. In that event the **MGraphic** layer should serve nicely. We strongly encourage you to use the **MGraphic** layer for all of your imaging needs.

Principle of Customization

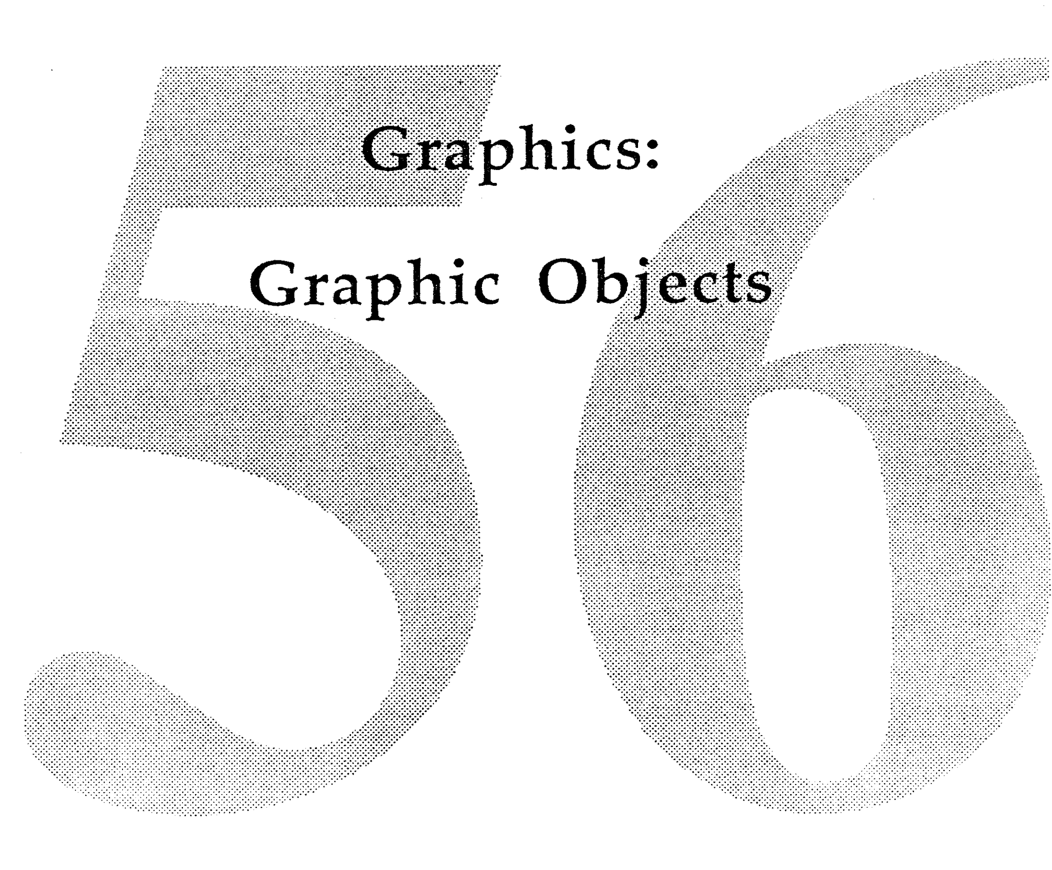
If you don't like using RGB colors, the wonders of object-oriented programming await you. Albert provides a variety of color model choices, probably more than anyone will ever use. And, so long as you relate your color space to one of ours, you can invent your own.

The same is true for most Albert style and paint objects. You can create your own objects within the parameters of what a GrafDevice expects an so customize the graphics process. Besides color spaces, you can add new line styles, new end caps, new joins, your own programmable shader, the works.

Principle of Extension

Of course, you may not want the works, because its a lot of work to completely redefine something. Take frame buffers, for example. You might want to implement your own curve representations. The path and area primitives of Albert allow you to extend their geometric data structures in arbitrary ways, provided you can relate the data in some way to the standard structures. In the event that your structure is imaged to a standard GrafDevice, the conventional information will be used. But, if your structure is imaged on your special, extended GrafDevice, you'll get whatever result you had in mind.

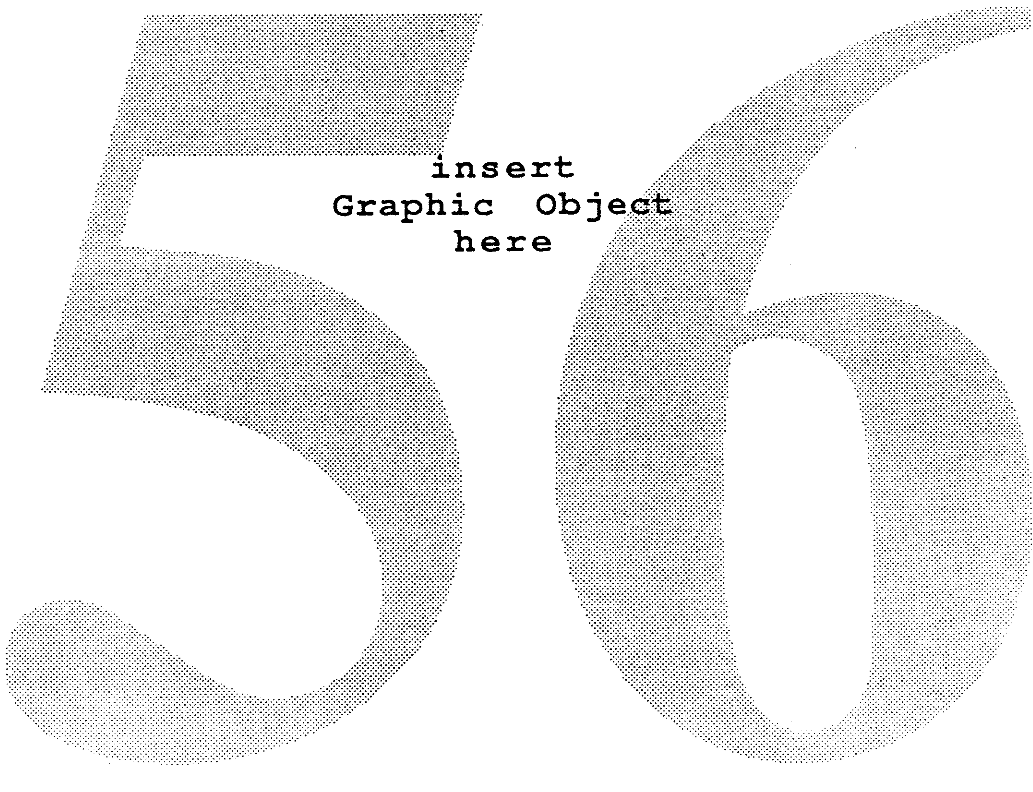
You don't have to implement an entire frame buffer to make this work. You simply have to derive a frame buffer from one of ours. In the render method for paths and areas you will first check the data structure to see if it is yours. If not, pass it up to the parent render method for processing. But if it is your special structure, you can image it in your own special way, or perhaps do a bit of processing and use standard calls to do the imaging. You get a new type of frame buffer for the cost of implementing part of a single routine.



Graphics:

Graphic Objects

56


insert
Graphic Object
here

56

Imaging With Graphic Objects [described by Roger Spreen]

As described under the *Principle of Inversion*, Albert supports the concept of a graphic object layer so that you can declare and render graphics in an intuitive, object-oriented fashion, irrespective of the device-centered layers underneath. This concept assures that your task of actually putting graphics on the screen is as simple as possible: usability is of prime importance.

It is important to understand that you can use any of the different "layers" of Albert and still achieve exactly the same results on your graphic device. These "layers" are simply sets of classes and methods which provide you with a slightly different paradigm for managing your graphics. For most standard graphics applications, the Graphic Object layer will maximize your convenience and minimize your development time in building an application.

The Components of a Graphic Object

No matter how you slice it, no matter which "layer" you use, there is still a certain amount of information you must provide in order to completely specify a graphic effect on your device. Albert chooses to divide this information into three separate components:

- Geometry: definitions of pure geometric shape, e.g. lines or rectangles.
- Transforms: arbitrary matrix transformations, most often used as affine transformations like translation, rotation, and scaling.
- Bundles: collections of appearance attributes, e.g. color, pattern, shading, joints and end caps.

This division of information leaves room for you to expand or customize portions of Albert, either in hardware or software, without your having to completely reinvent an entire graphics system. The Graphic Object layer organizes these three components together into easily manageable classes, where, in general, each class represents a different geometric type.

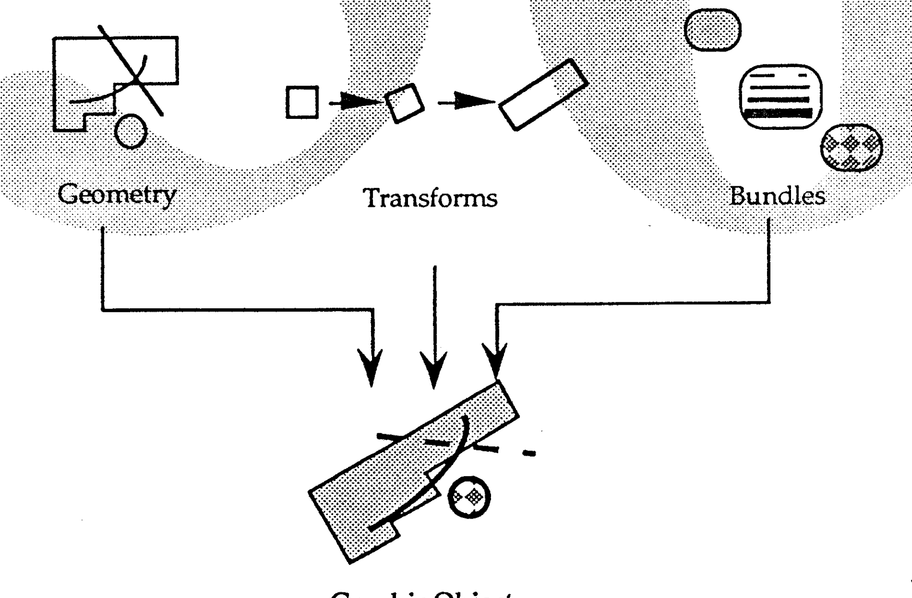


Figure 1. Components of a Graphic Object class

Advantages of the Graphic Objects

Primarily, the Graphic Objects offer you convenience. Albert can leverage object-oriented programming to encapsulate the three components together, eliminating the need for you to manage all the pieces yourself. Furthermore, just by forming a simple collection of Graphic Objects, you can have "Display List" functionality which is still device-independent. (For convenience, the Graphic Object class includes two built-in collections: a simple grouping object and a hierarchical grouping object.) Thus, a collection of Graphic Objects can serve a similar purpose as QuickDraw's "Picture." Similarly, the Graphic Objects can also be used to transport graphics from one application to another. One additional advantage of using a collection of Graphic Objects as a clipboard medium is that it can easily be edited.

The generic nature of the objects in this layer means that you can write routines that handle Graphic Objects without knowing their internal details. Existing applications that handle Graphic Objects generically (such as in a display list, or from the clipboard) will automatically be able to handle new, custom Graphic Objects that are placed in the system.

Finally, the Graphic Object layer supports both 2D and 3D objects. Thus, existing applications will be able to display 3D graphics that are imported as any other clipboard object. Also, the use of 3D graphics will be more intuitive to the developer who is new to 3D graphics, but is already familiar with Albert's 2D capabilities.

Taxonomy of the Graphic Object Layer

The Graphic Objects are distinguished by their inheritance from the MGraphic class, which gives them their displayable functionality and also provides basic Pink functionality (due to its MPersistent --> MCollectible ancestry). This is also why the set of Graphic Object classes are often referred to as "the MGraphic classes," and any single Graphic Object is often referred to as "an MGraphic."

Because the Graphic Object classes' main purpose is to encapsulate geometry, most of the basic Graphic Object classes also inherit from Albert's geometric classes. The only exceptions are collection classes and Images (Pixmaps). See the chapter on "Geometry" for more details.

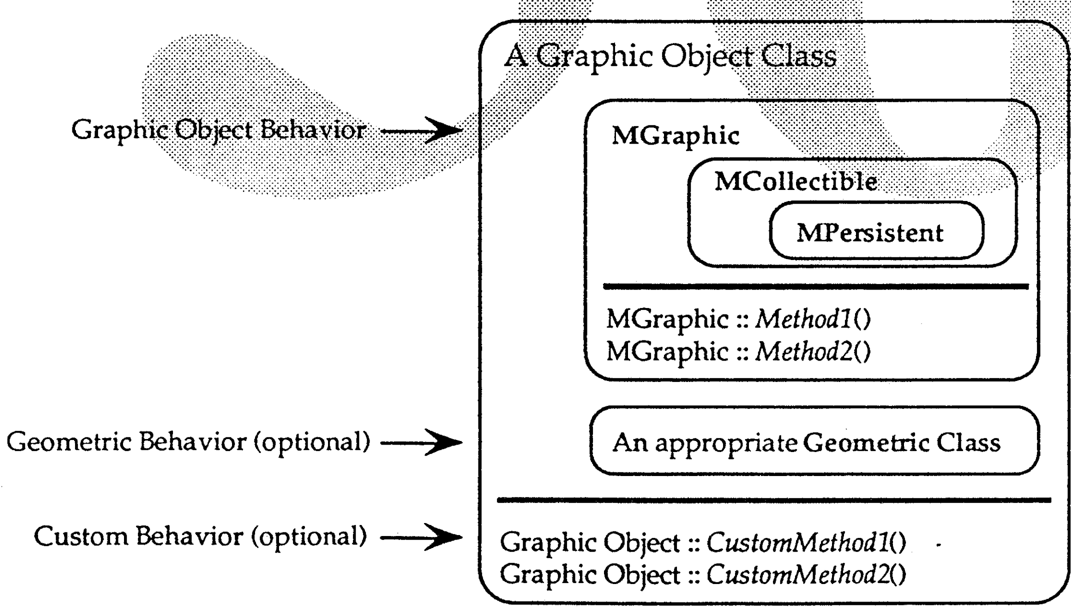


Figure 2. Graphic Object class structure

The complete set of Graphic Object classes is listed below (along with the geometric class it is derived from, if any). Note that there is no Graphic Object point, since Albert does not render "points" *per se;* depending on the resolution and characteristics of the output device, a "point" could be of arbitrary size and shape. Is a "point" square, rectangular, oval, or round? Thus, to render what we think of as a "point" or "marker," it is necessary to render some specific geometric shape at a specific, device-independent size. However, it is obvious that "points" are a useful concept in a graphics system, so throughout Albert, and more specifically, throughout the Graphic Object classes, the Geometric class **TGPoint** is used for specifying coordinate data.

Line-oriented 2D classes

- **TLine:** A line segment formed by 2 endpoints (**TGLine**).
- **TArc:** An arc defined by 3 points (**TGArc**).
- **TCurve:** A general purpose approximating spline curve; depending on the number of control points given, this can be a Quadratic Bézier curve, Cubic Bézier curve, or a NURB spline (**TGCurve**).
- **TPolyLine:** A set of connected line segments (**TGPolyLine**).
- **TPath:** An arbitrary collection of connected 2D line-oriented geometries (**TGPath**).

Area-enclosing 2D classes

- **TRect:** Your classic rectangle, defined by the upper-left and lower-right corner points (**TGRect**).
- **TRoundRect:** A rectangle with rounded corners (**TGRoundRect**).
- **TEllipse:** Circles and ellipses (**TGEllipse**).
- **TPolygon:** A polygon with an arbitrary number of 2D vertices (**TGPolygon**).
- **TGlyphRun:** A set of text characters, defined as character codes (or "glyph codes" which index some outline font description (**TGGlyphRun**).
- **TArea:** An arbitrary collection of other area-enclosing 2D objects (**TGArea**).

Line-oriented 3D classes

- **TLine3D:** A line segment formed by two 3D endpoints (**TGLine3D**).
- **TPolyLine3D:** A set of connected 3D line segments (**TGPolyLine3D**).
- **TPath3D :** (TBD)

Surface-oriented 3D classes

- **TBox3D:** A right rectangular prism (**TGBox3D**).
- **TPolygon3D:** A polygon with an arbitrary number of 3D vertices. (**TGPolyNet3D**).
- **TPolyMesh3D:** A set of connected, 4-sided polygons formed by 3D vertices that topologically form an array, thus yielding a "patchwork quilt" of polygons (**TGPolyMesh3D**).
- **TPolyNet3D:** A set of connected polygons formed by an arbitrary number of 3D vertices (**TGPolyNet3D**).
- **TSurface3D:** An approximating-spline surface formed by a mesh of 3D control vertices (**TGSurface3D**).

Image classes

- **TImage:** A 2-dimensional array of pixel values, a.k.a. "Pixmap" (no geometric equivalent).

Graphic Collection Classes

- **TGroup:** A simple list of other Graphic Object objects.
- **TInstance:** A reference to a single Graphic Object object, along with a unique transformation matrix.

Graphic Object Methods

By virtue of descending from the **MCollectible** and **MPersistent** classes, a Graphic Object can be used with the various collection classes, and it can be "streamed" to provide a basic storage, retrieval, and communication mechanism. By virtue of being a descendant of **MGraphic**, a Graphic Object gains 4 additional capabilities: Rendering, Transformations, Bundles, and Hit Detection.

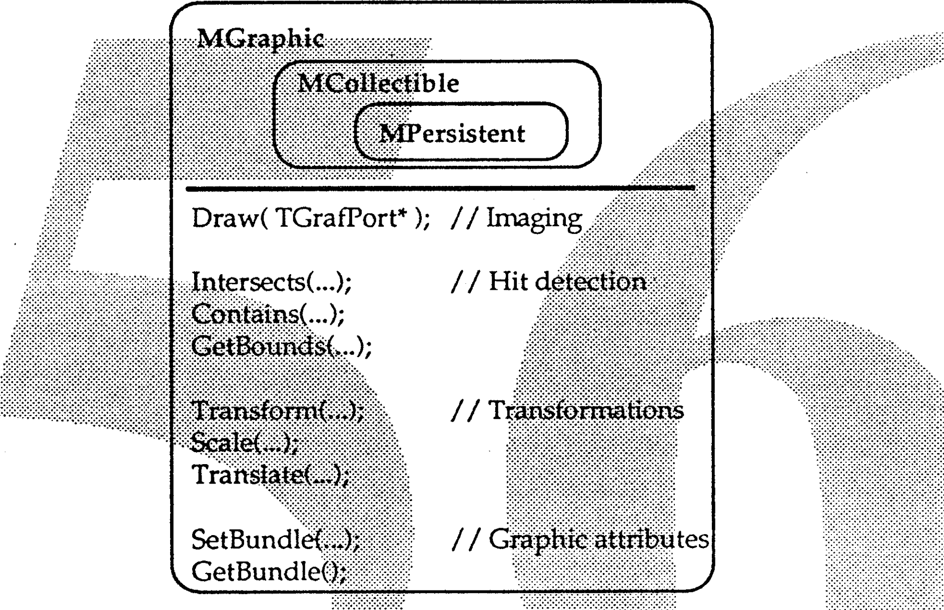


Figure 3. The MGraphic class

Rendering

The key method of a Graphic Object, whether 2D or 3D, is "`Draw( TGrafPort* port )`"; thus, you simply tell the object where to draw itself. All the remaining information need by Albert is contained within the Graphic Object.

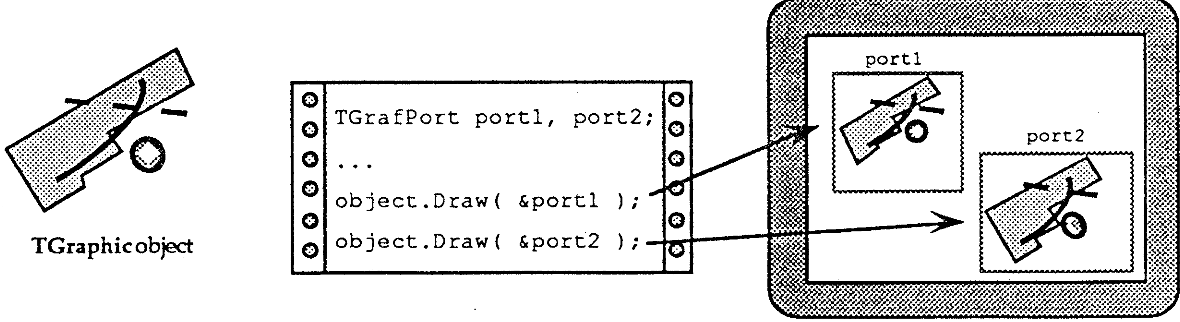


Figure 4. Rendering a Graphic Object

Since a Graphic Object and a GrafPort both contain transformations and bundles, it is important to understand the interactions between the two during the "`Draw()`" process: the Graphic Object sends its geometry into the GrafPort to be rendered there. If the Graphic Object has a bundle, it sends *that* into the GrafPort, and its values take precedence over those of the GrafPort's bundle. The GrafPort's bundle is used only to provide defaults for values that the object's bundle does not specify. Thus, if you leave values unspecified in an object's bundle, you are relying on whomever sets up the GrafPort's bundle values to "do the right thing." The Graphic Object's geometry is additionally transformed by the GrafPort's transform (via post-concatenation). This makes it simple to transform all objects in a GrafPort, so that a GrafPort can, for example, translate everything rendered inside itself to achieve a scrolling effect, or for another example, scale everything to achieve a zoom-in effect.

This generic "Draw" ability makes it simple to handle all Graphic Objects generically. For example, an application can simply keep a list of all such objects, and can render them all with a simple loop:

```
void RenderAllGraphics( MGraphic* objects[], int cnt,
                                            TGrafPort* port )
{
 for ( int i = 0; i < cnt; ++i )
     objects[i]->Draw( port );
}
```

Figure 5. Rendering a display list

Transformations

Every Graphic Object supports a "`Transform()`" method, which effectively transforms every point in the object through the given **TMatrix**. Note that some of the Graphic Objects implement this by carrying a transformation matrix that is only applied at rendering time, while others immediately transform the actual points that form the geometry. The inconsistency of the implementations is still an architectural issue under discussion.

```
MGraphic::Transform( const TMatrix& );
MGraphic::Transform3D( const TMatrix3D& );

MGraphic::Translate( const TGPoint& );
MGraphic::Scale( const TGPoint& scale, const TGPoint& centroid );

MGraphic::Translate3D( const TGPoint3D& );
MGraphic::Scale3D( const TGPoint3D& scale );
```

Figure 6. Transformation methods

A **TMatrix** represents only a 3x3 transformation matrix, which is insufficient for 3D graphics. Thus, the **TMatrix3D** supports a full 4x4 transformation. There is no danger in calling "`Transform()`" on a 3D object, or "`Transform3D()`" on a 2D object; matrix elements will be discarded or added as necessary. For details on creating transformation matrices, see the appropriate chapter.

Since most transformations tend to perform translation and scaling, the **MGraphic** class supports these methods explicitly, which saves the trouble of having to work with matrices. With "`Scale()`," the first **TGPoint** gives the X and Y scaling, and the 2nd **TGPoint** indicates the point about which the

scaling should occur. Just as with the "Transform()" method, there exist equivalent 3D forms of Translate and Scale (and again, it is safe to mix these calls with 2D graphics or vice-versa):

Bundles

Because a bundle is simply placed inside it, the MGraphic class only needs to provide "Get" and "Set" methods. For details on creating and modifying bundles, see the appropriate chapter.

```
void MGraphic::SetBundle( TBundle* );
TObjectBundle*  MGraphic::GetBundle();
```

Figure 7. Bundle methods

Hit Detection

Every Graphic Object can aid in by returning geometric information about the object in its rendered state. There are 3 flavors of geometric information: bounding areas and volumes, intersections, and containment.

```
TGRect&   MGraphic::GetBounds();
TGBox3D&  MGraphic::GetBounds3D();

Boolean   MGraphic::Intersects( const TGRect& );

Boolean   MGraphic::Contains( const TGRect& );
```

Figure 8. Hit Detection methods

"GetBounds()" returns the smallest coordinate-axes-aligned 2D rectangle that completely encloses the controlling points of an object. Note that for approximating curves, the control points used in the GetBounds() calculation are not points that are on the curve. Also, the pen-widths of the Graphic Object's Bundle *are* used where appropriate in this calculation. (For 3D objects, GetBounds() returns the bounding rectangle that represents only the Z-axis projection; i.e., it gets the (X,Y,Z) bounds and then eliminates the Z value.). "GetBounds3D()" returns the smallest 3D Box (aligned along the coordinate axes) that completely encloses the object.

"Intersects()" indicates whether the given rectangle intersects any point on the Graphic Object. (For area-enclosing Graphic Objects, this includes any point inside the area.)

"Contains()" indicates whether the given rectangle is completely contained by the Graphic Object. (Note that "Contains" has not yet been implemented for line-oriented Graphic Objects.)

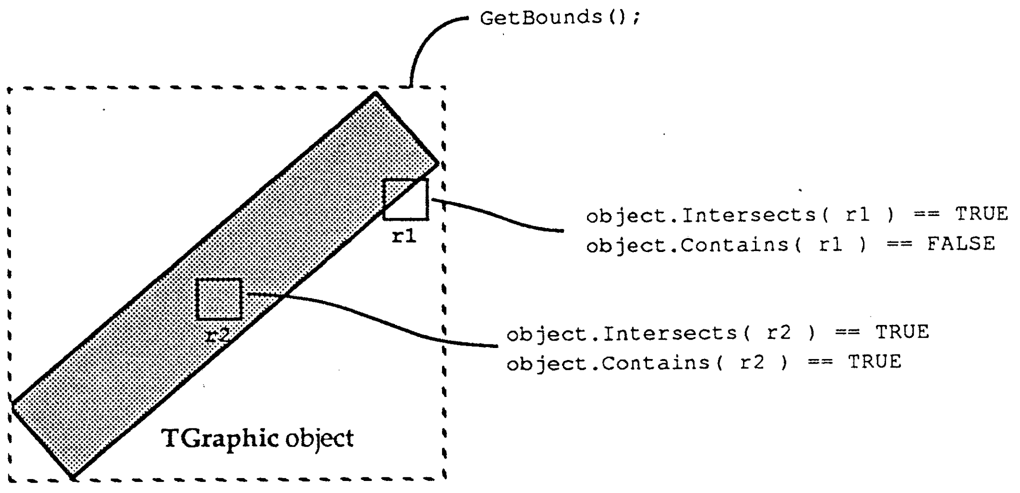


Figure 9. Hit Detection results

Certainly the most common use of the Hit Detection methods is picking objects with the cursor. Picking an object with a cursor could be implemented like this:

```
MGraphic* object;
GCoordinate cursorX, cursorY;

// Get the cursor location somehow

TGRect cursorLocation( cursorX, cursorY, cursorX, cursorY );
if (object->Intersects( cursorLocation ) == TRUE)
{
   // cursor was over object!
}
```

Figure 10. A sample picking routine

Text: the TGlyphRun class

Text at the Albert level means simply the painting of characters on the screen, with no editing or fancy layout facilities. Actually, with the ability to paint International text and symbols, the term "character" is inappropriate. What is normally thought of as a "character" may actually be made up of several distinctly separate pieces of area-enclosing geometry, called glyphs.

For the purpose of rendering text, Albert treats glyphs as any other piece of area-enclosing geometry. Since text is most commonly displayed as a string of glyphs, Albert supports a collection of these glyphs in a "run of glyphs," or "glyph run." The geometric class is **TGGlyphRun**, and following the normal Albert naming convention, the Graphic Object class is a **TGlyphRun**. As with any Graphic Object, it has a Bundle that determines its colors, etc., and it is rendered simply with "`Draw()`." For more details on this class and on Font Management, see the appropriate chapter.

The TGroup Collection

Albert supports two commonly-used types of "display list" collections; by making these collections Graphic Objects themselves, they can be rendered, hit-tested, or transformed in exactly the same ways as described above.

The **TGroup** class is a simple array-style list of graphic objects. It offers methods for inserting and removing items from the list, much like the functionality of the underlying Pink Toolbox TDeque classes. The more complex methods use a **TGroupIterator** facility.

```
// Adding to the list

void        Add( MGraphic* newObject );
void        AddFirst( MGraphic* newObject );
void        AddLast( MGraphic* newObject );

// Removing from the list

MGraphic*   Remove( const MGraphic* obj );
void        RemoveAll();
MGraphic*   RemoveFirst(); // returns the removed item
MGraphic*   RemoveLast();  // returns the removed item

// Reading through the list

            TGroupIterator( const TGroup* );
MGraphic*   TGroupIterator::First();
MGraphic*   TGroupIterator::Next();
MGraphic*   TGroupIterator::Last();
```

Figure 11. **TGroup** list-editing methods

Since the TGroup is itself a Graphic Object, it can contain other TGroups; thus it can form a hierarchical display list.

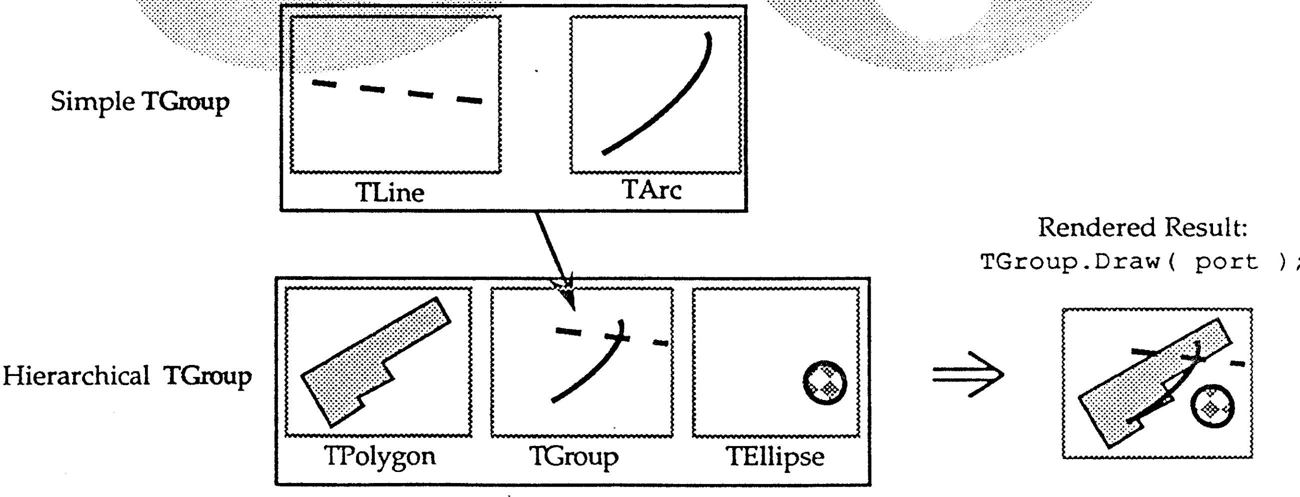


Figure 12. Using **TGroups** to make a hierarchical display list

TGroup Methods

In general, a **TGroup** method recursively calls the same method of all the children. Thus, if one **TGroup** contains another, the hierarchy will be traversed in a depth-first search manner.

- `GetBounds`: recursively gets the bounding box of all control points from its children, and returns the bounding box that encloses all the children's bounds.
- `Intersects`: recursively calls Intersect for each child, and returns TRUE as soon as any child reports TRUE.
- `Contains`: gets the bounding box of the **TGroup** (see `GetBounds`), and returns TRUE if the given rectangle is contained.
- `Scale, Translate, Transform`: recursively calls the appropriate method of each child.
- `Draw`: recursively calls each child's `Draw` method..

The **TGroup** has its own bundle, and supports the standard "Get/Set" Bundle operations to modify it. It is important to understand the use of this bundle during rendering, especially when used with hierarchical groups. Essentially, the **TGroup**'s bundle is only used to provide default values where the children's bundles have not provided more specific values. For example, if the **TGroup**'s bundle specifies red fill paint, and the child specifies blue fill paint, then the child will be filled in blue. However, if the child has no fill paint, it will be filled in red. This process is accomplished by the **TGroup** putting its bundle into the GrafPort before rendering any of the children; as mentioned previously, the GrafPort uses its bundle to fill in any defaults. When the **TGroup** finishes rendering all the children, it restores the GrafPort's bundle to the original state. Note that if one of the children changes the GrafPort's bundle, all the other children in the **TGroup** could be affected by that change, until the **TGroup** resets the GrafPort's original bundle.

Instancing

The **TGroup** provides both a hierarchical display list ability and "instancing" functionality; that is, it can point to the same child object multiple times. When that **TGroup** is rendered with Draw(), it will render that child repeatedly, as many times as it appears in the list. Unfortunately, that child will be rendered repeatedly at the same location, on top of itself, over and over; after all, there's nothing that distinguishes one reference to that child from another.

Thus, an additional class is offered that can distinguish the multiple references: the **TInstance** class. A **TInstance** is a collection class that holds only one child, but also includes a transformation matrix (**TMatrix**) that is applied to the child when rendering it. This is performed by:

1. Pushing the GrafPort's internal matrix stack to save the current matrix.
2. Pre-concatenating the **TInstance's** matrix onto the GrafPort's matrix.
3. Rendering the child (which may be a **TGroup** or another **TInstance**!).
4. Popping the GrafPort's matrix stack to restore the previous matrix.

Notice that by using the matrix stack, it allows recursive rendering of **TGroups** and **TInstances**. You only need to worry about the appropriate transform in the specific **TInstance**, and the rendering process of `Draw()` will manage the recursion.

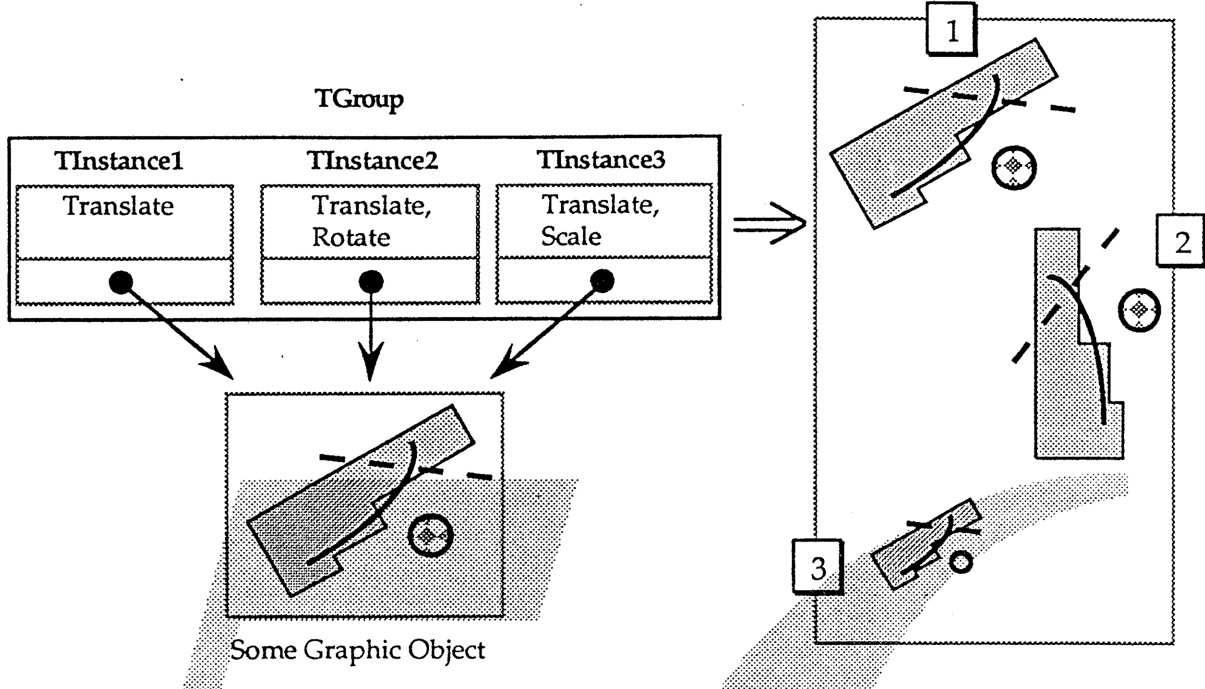


Figure 13. A TGroup with 3 instances

TImage

With new devices like scanners, film recorders, color printers, and full color frame buffers, and with a growing emphasis on "Multi-media" abilities, image processing is becoming an increasingly significant Macintosh feature. However, the current Macintosh's support for storage, retrieval, and display of images is very clumsy, leading to inefficient and inconsistent schemes between applications. Applications tend to be tightly tied to a specific image file format and a specific graphics device at a specific resolution.

Because an "Image," or "Pixmap," is a discrete collection of color samples, it is inherently resolution dependent; however, there are many different image-processing algorithms for manipulating images between different resolutions and sampling parameters. Albert provides several basic facilities such that an Image can be displayed on any device in the system in a manner that is transparent to you. By default, you need not worry about the resolution or color palette of a specific device; Albert simply displays the image at the highest quality the device can achieve, using whatever color matching techniques are necessary. Shrinking, stretching, rotation, warping, etc., are all functions which Albert will also support in order to make image processing more accessible to developers.

Albert supports images with the TImage class, which is just another Graphic Object. Thus, images can be rendered with the standard "Draw()" method. However, because of the potential complexity of an image, there are a number of additional parameters and concepts required to set up a TImage; for more details, see the appropriate chapter.

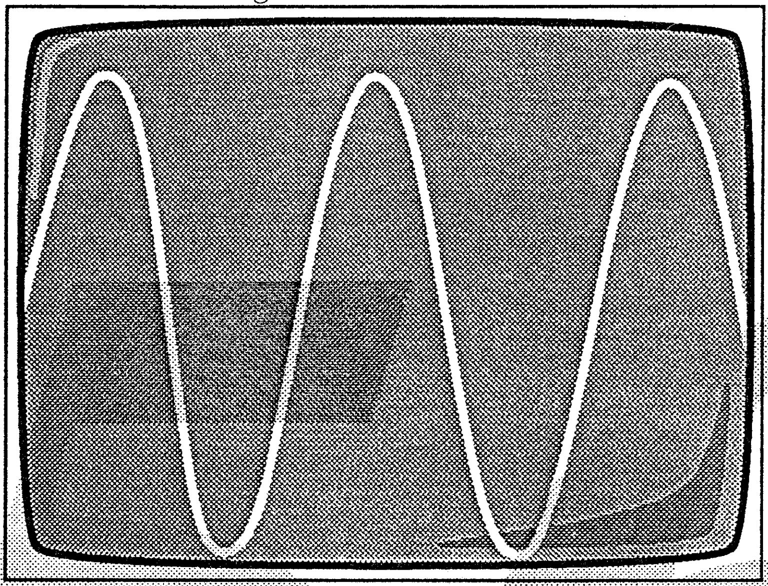
The Outer Limits

Printing and Image Management

56

The Outer Limits

Printing Architecture Overview



"There is nothing wrong with your screen. Do not attempt to adjust either the image or the printer. *We* are controlling the process. *We* will control the horizontal. *We* will control the vertical. *We* can clip the output. *We* can scale the result. *We* can change the focus to a soft blur, or sharpen it to crystal clarity. For the next hour, sit quietly and we will control all that you see and hear. We repeat: There is nothing wrong with your computer or printer. You are about to experience the awe and mystery which reaches from the inner bits to the outer hardcopy. PLEASE STAND BY..."

—Adapted from "The Outer Limits" television series.

Bayles Holt
Ryoji Watanabe
Jay Patel
Mahi deSilva

56

Table of Contents

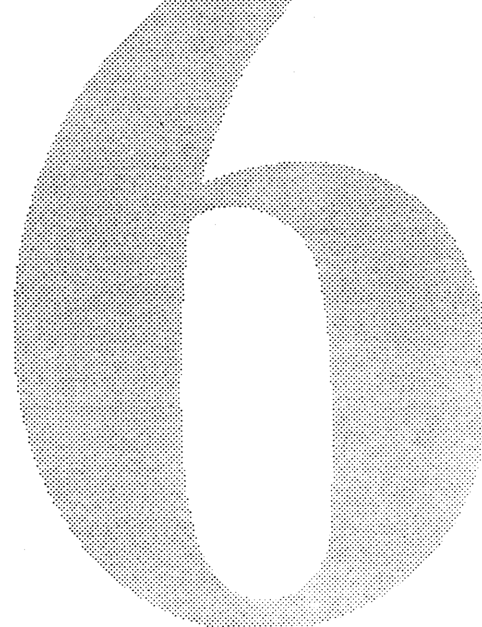
Printing and Imaging Architecture Overview

The Outer Limits........ii
Table of Contents........iii
Table of Figures........iv
I. Introduction........1
  Goals........1
II. Architecture and Overview........2
  Printing Services........2
  Color........2
  Scanning........4
    Scanning Model........4
    Scan Devices........4
    Scan Image Links........5
  Printing........5
    The Printing Model........5
    User Interface........6
    Printing Devices........6
    Printing Classes........7
    The Printing Process........8
    The Print Server........9
    Printer Teams........9
III. What It Does and How It Does It........9
  Printing Services........10
    Creating Documents and Assigning Printers........10
    Page Setup, Document Format and Metrics........10
    Dynamic Print Properties........10
    Printer Selection........10
    Managing Print Jobs........11
    Other Services........11
  Multimedia........11
  User Interface Samples........12
IV. Print Classes and Objects........12
  MPrintable........12
  Default Printing........13
    General Document Model........13
    Spreadsheet Model........14
    Stack Model........15
    Pile Model........16
    Other Document Models........16
  TPrintJob........16
    Support Classes........18
      Media Classes........18
        Trays and Bins........19
      Printer Classes........23
        Printer Addresses........23
        Printer Attributes........23
        Device Access........24
        Logical Printer........24
V. Examples........24
  Setting up a Printable MGraphic........24
    Paging........25
    Printing........25

56

Table of Figures

Figure 1. The Printing architecture is specifically targeted for future systems, not the least of which is Jaguar....1
Figure 2. An example of the color model....3
Figure 3. All the components of printing....6
Figure 4. An MPrintable....7
Figure 5. A TPrintJob....7
Figure 6. The MPrintable class....12
Figure 7. A generic document model....14
Figure 8. The Spreadsheet Model....15
Figure 9. The stack model....15
Figure 10. The TPileModel....16
Figure 11. The TPrintJob class....16
Figure 12. TRecordingGrafDevice....18
Figure 13. Possible media orientations....20
Figure 14. Stacking order defined....21
Figure 15. Duplex printing with various kinds of media....22
Figure 16. The Printer Class....23



56

I. Introduction

This document gives a brief overview of the printing architecture. Printing, as far as this architecture is concerned, also includes scanning, plus other forms of multi-media, specifically video, cameras, frame grabbers, film recorders, plotters, and VCRs. In addition, because animation is a major component of the Pink environment, the Printing architecture provides some facilities for printing frames in an animated sequence and for synchronizing sound with the animation. The printing architecture is viewed as a low level platform for video, multi-media and animation uses.

The architecture is designed to be simple and easy for novice users, but also flexible and extensible enough to provide facilities for the most sophisticated application.

All imaging within the Pink environment is device-independent, that is, the graphic model and imaging system have no default or built-in resolution, and no assumptions are made about the resolution of any object created in it[1]. What this really means is that any printing job initially intended for one target device may be redirected to another target without causing gratuitous reformatting or similar repercussions in the document.

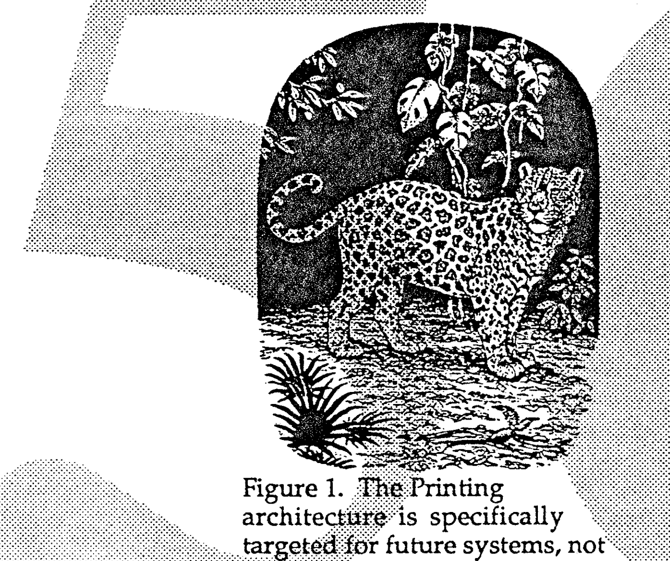


Figure 1. The Printing architecture is specifically targeted for future systems, not the least of which is Jaguar.

Goals

The major goals regarding printing in the new system are to produce a printing model that surpasses the capabilities found in any other environment including the original Macintosh. Known restrictions, that have plagued other models, have been removed and replaced with a more flexible model that allows expansion and growth, and adds additional facilities that make development easier, faster, and more secure, while at the same time providing consistency, device independence, and ease of use.

In summary, some of the services provided by the Printing and Imaging architecture are:

- Inclusion of multi-media and scanning facilities in the basic architecture.

[1] Objects such asTImages may have a specified sampling resolution, but this is a different animal. There is no overall implicit resolution in the graphic model.

- Support for other non-standard devices such as plotters, video recorders, and milling machines.
- Automatic, high quality, device specific, 'WYSIWYG', including color matching.
- User and application parameterization of all printing and imaging functions.
- Automatic, overloadable device control for each printer, such as page ordering, bin selection, and so forth.
- Variable page sizes within a single document and support for custom paper sizes.
- Arbitrary page ranges with automatic collation.
- Automatic (client modifiable) printer specific optimizations.
- User alterable queues for background printing and remote spooling for all devices.
- Access to accounting and use-tracking as might be needed in specific printer installations.
- Provide sub-document page reordering, collation, and job redirection, both before and after a job has been spooled.

The rest of this document will provide a basic overview of an Imaging and Printing architecture and present the classes that implement the architecture. We will describe how these various classes operate and interact with each other and give some examples as to how they should be used.

II. Architecture and Overview

Printing Services

There are four major services provided by the print architecture. These are:

- Provide for choosing among printers
- Provide page set up and page metric information
- Print
- Manage print jobs

Many of these services may be controlled jointly by the user and the client application with the application determining how much control the user may have. All of these services have a user interface and a standard set of dialogs provided by the system. The application determines to what extent the standard or default interface is presented to users. It has the option of presenting the given user interface without modification, adding to it by including its own, or completely replacing the interface to suit its own particular needs.

Color

At the core of the printing architecture is the graphics model and at the core of the graphics model is the color model. The graphics model is documented heavily elsewhere but we will say a few words about color here.

While true perceptual color fidelity may not be truly achieveable without full environmental control, the imaging architecture does attempt to provide some degree of color consistency through color matching between devices. The color model is ultimately based on a central CIE coordinate color space, internal to the system, into which all devices may be mapped. Each device connected to the system is characterized by its color gamut and a transformation map that specifies the range of color available on the device. Any color data being passed between a device and the system is mapped from one to the other through this transformation. In this manner, any two devices can be color matched by at most two transformations, one to get from one device to the system color space and the other to get from the system color space to the second device.

A number of alternative color spaces are also supported such as the well known RGB, HSV, HLS, CMY, Luv, and so forth. Some of these spaces, without a formal color specification (like RGB) simply use the intuitive approach. There are also a standard set of transformations available to provide conversion from one color space to another.

The printing architecture is augmented by this color model and relies on it for color matched printing. Although color processing is provided by default, clients are allowed to bypass the color correction and transformation steps if it best serves the purposes of the application.

If all devices have a fully characterized color space, then transformation between devices can be achieved as in the following diagram. Questions regarding gamut mapping, white point setting and so forth can be parameterized and dictated by clients.

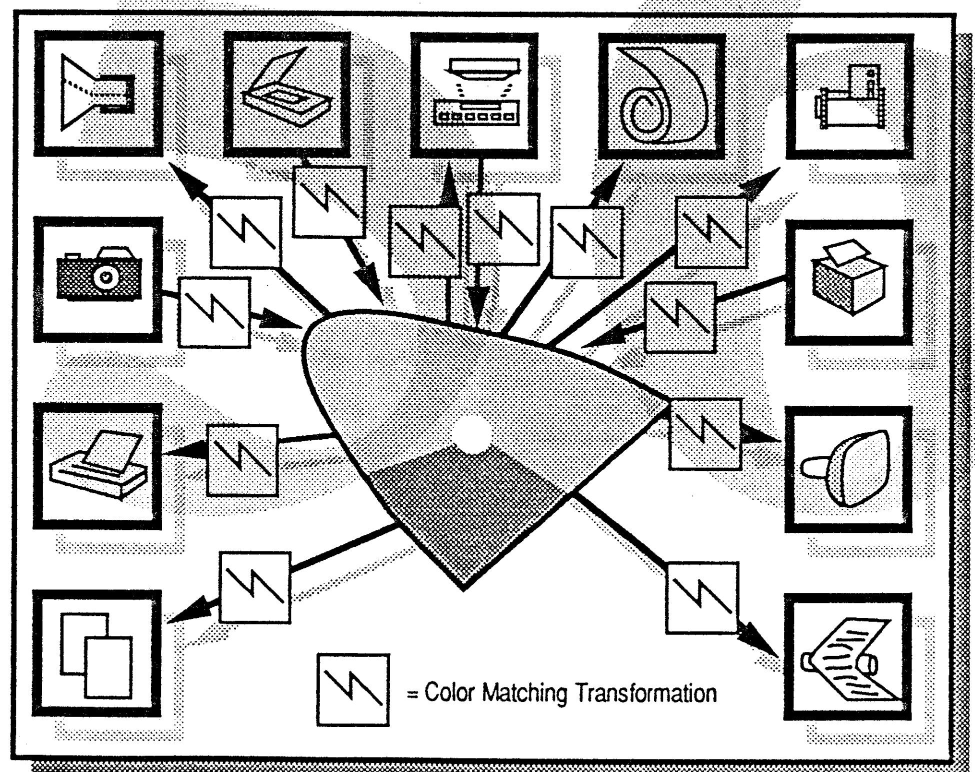


Figure 2. An example of the color model used by the printing and scanning architecture.

Now that we have examined the color model, lets look at two other major components of the printing and imaging architecture, scanning and printing. We will present a brief summary of scanning, but the primary focus of the rest of this document will be printing.

Scanning[2]

Scanning is defined as the process of inserting information into a document from some source external to the processor that is not generated directly by the user. Input data can be in the form of images, characters, video, Albert Graphics or any collection of these.

Scanning Model

Conceptually, scanning is simply the process of adding content obtained from an external scanner to a document. The data of interest may optionally be saved in secondary storage for later use as well. Scanned information may be processed or filtered before it is placed in a document if desired. The filtering and processing step may be augmented by the application, by third parties or the user herself.

The types of preprocessing available might include image processing , compression, character recognition, feature extraction, and so forth. There is always a color matching filter available which allows the color gamut of the input device to be mapped to the internal color space. The user may choose to ignore color matching if that is her wish.

Other filters such as those that perform halftoning or dithering also usually produce unsatisfactory results and are not recommended for input scanning use. Halftoning and dithering are highly device and resolution dependent and their effects are very difficult to remove. The printing system can provide any method of halftoning necessary and can produce much better quality if it is left to the printing process. Special effects to produce patterns and other visual effects may of course be added[3], but these should not be produced by premature halftoning.

When images are scanned they are typically done at the highest available resolution and pixel depth in order to provide the highest quality for the user. The level of quality, however, can be controlled by the application or the user.

Scan Devices

Virtually any type of device that provides image information can be supported, for example, video cameras, flat bed and slide scanners, OCR scanners, or hand held devices.

Like printers, scanning devices are user selectable, but unlike printing, several different scanners may be chosen for a single page at the same time.

---

[2] All the details about the scanning mechanism are described in a separate section entitled "Scanning".

[3] Special patterns such as 45° lines, full page circular rings, diamonds, proprietary dots and so forth can all be added directly to the image iself. The results of this operation when printed are much more favorable and the whole mechanism is more device independent.

Scan Image Links

Because input images typically run in very large sizes (up to 240 MegaBytes or more depending on size, resolution and quality requirements), input fragments are not stored more than once. Images are not copied in totality into every place they occur, rather an image reference or tag is attached to the relevant data before it is attached to the document or application where it is used. With printing in particular, when images are encountered in a document being printed, a link[4] to the image in question is placed in the spool file rather than including the entire image itself.

Printing

Now let's talk about the Printing model.

All printing in the Pink environment is deferred. This simply means that print jobs are placed in imtermediate storage before they are actually printed. The purpose of deferred printing is to unburden the application from having to be tied to slow printers and to allow Printer Teams to process pages at leisure. Deferred printing is not equivalent to slow printing; a more accurate metaphor is a pipeline. We will describe the implementation of the deferred printing model.

The Printing Model

There are three Teams associated with printing: the document or application, the Print Server, and the Printer Team. We will refer to all three of these in subsequent discussions, though the Printing Classes used in the document by the application will be the primary focus for the rest of this paper. The Print Server and Printer Team will be presented briefly and referred to often but are documented more thoroughly in separate sections. The following diagram shows the relationship of the three teams.

56

[4] These links should not be confused with shared document links as they are not the same thing. The links described here are more low-level and invisible to the ordinary user.

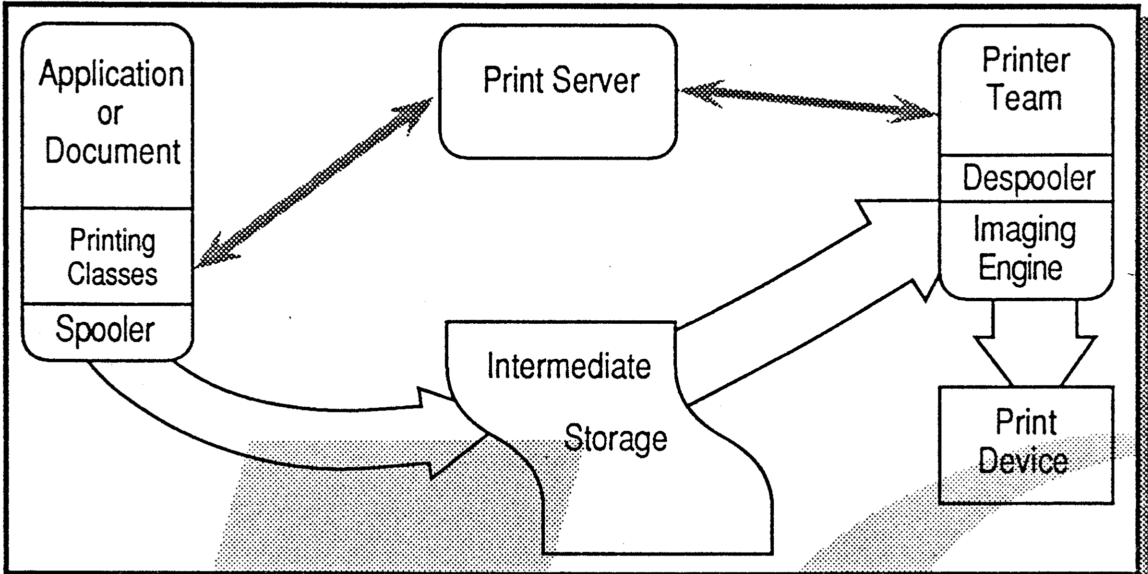


Figure 3. All the components of printing. The Printing Classes, through which printing is accomplished, are accessed by developers. The Print Server oversees the whole process and the Printer Team takes care of printing on a specific printer. The gray lines represent control information while the hollow arrows indicate the flow of printing data.

Here is a brief definition of each of these three entities. The Printing Classes encapsulate the entire printing process from the application perspective. We'll talk more about the classes throughout this paper. Applications interface with printing almost entirely through the Printing Classes and have little to do with the operation of the Print Server and Printer Team both of which function in the background independently from the application.

The Print Server can be thought of as an overseer, who manages all print jobs and and all printers. The Printer Team is a device specific print driver that is given a print job by the Print Server and then prints it on a given printer.

User Interface

There are several user interface dialogs provided by the print architecture although their use, though recommended, is not enforced. These interfaces provide a document creation or PageSetup dialog, a Print dialog, a PrinterSelection dialog, and a JobQueue dialog. These dialogs are minimal and have very few built-in functions already in them but the first two may be augmented by the application if desired. All are necessarily device dependent.

There will be more about the user interface to printing as the Pink Human Interfaces become more mature and well defined.

Printing Devices

We must say a word about Print Devices. Graphic input-output devices may be divided into two categories, Primary and Secondary. The deferred printing model is intended for Secondary types of devices as defined below. Primary devices are handled more directly and usually by other means than printing.

Primary devices are:

- Always there, that is, always attached to the machine, for example the monitor.

- Fast and responsive and are most likely used interactively.
- Typically "near", meaning within the client's own address space.

Secondary devices are:

- Intermittent or dynamically connected; not always on-line.
- There may be many such devices from which one or two are chosen periodically by the user or application.
- May be slow and probably not used interactively.
- Okay to be "far", or outside the client's address space in other processes or teams, maybe even physically remote.

The deferred printing model can be thought of as a way of wrapping up a Primary device and converting it into a Secondary device. This will become more clear as we talk about Albert Graphics and Printing Classes.

Printing Classes

There are two principle classes that the application needs to know about for printing. These two classes, MPrintable and TPrintJob, are shown in the following figures. We will make use of an Albert metaphor in explaining these classes and how they are used.

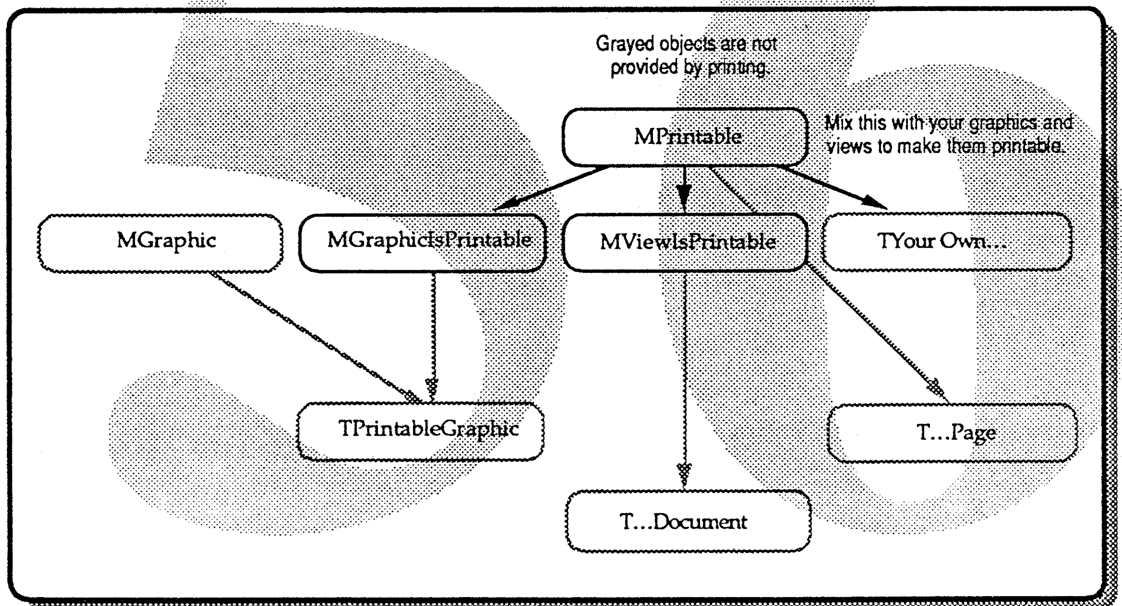


Figure 4. An MPrintable represents something that can be printed. It can be mixed with anything that can be drawn in order to make it printable.

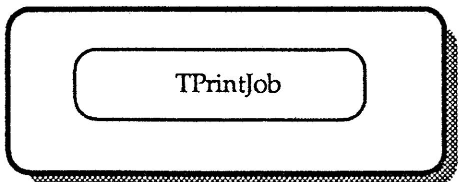


Figure 5. A TPrintJob contains everything about a printing session that a document needs to print. Like a TGrafPort, it contains the state of printing in bundles called TPrintSettings and TJobSettings, somewhat like TBundles in Albert Graphics.

We assume that the reader is familiar with Albert Graphics to some extent because we will draw an analogy between the graphic and printing interfaces:

In Albert, an MGraphic is something that knows how to draw. In printing, there is an equivalent class that knows how to print called MPrintable.

In Albert, a TGrafPort is the object through which all drawing operations are sent. In printing, a TPrintJob is is the object through which all printing operations are sent.

In graphics an MGraphic is "drawn" into a TGrafPort, in printing an MPrintable is "printed" into a TPrintJob.

It should be noted that this analogy should not be taken too literally. It is not necessary, for example, to mix MPrintable with every object you ever intend to print. The metaphor is to show how they are to be used.

A TPrintJob represents the encapsulation of a complete printing session for a document or portion thereof. To print, you create a TPrintJob and "print into it". An MPrintable is something that knows how to print itself into a given TPrintJob.

It is possible for any application to have any number of TPrintJob objects being used simultaneously. It is also possible two "mix" two different documents into one TPrintJob, though they will emerge at the printer as a contiguous stack. The function of a TPrintJob is to delimit a contiguous run of pages.

The Printing Process

The TPrintJob may be viewed as something that exists to "wrap up" a TGrafPort (designed as a Primary device channel) to convert it into a Secondary device channel. This scheme can be used for Primary and Secondary devices in general, for example those not necessarily dealing with either printing or graphics.

When an application is prepared to print, it creates a TPrintJob and tells MPrintable to print into it. It is a good idea for the application to invoke a separate background team to do this, although nothing in the Printing Classes depends on this[5]. The application printing team may be autonomous to the main application body in the sense that it can exist independently of and separately from the editing session with the user. The major advantage for doing this is that the application is then free to quit or perform other functions, and the printing team may be used for printing documents from the finder without involving the entire application.

During the printing process, as pages are drawn by an application, the data being created is spooled into an intermediate file, or piped directly through memory, to a specific Printer Team. The application is not aware of what actually happens to the data. This is usually of no consequence to the application except that a printable object's drawing routines may be called more than once during the printing session. If this is a problem for the application and the application is not able to handle multiple drawing calls without undue burden, it may create a TPrintJob that only calls drawing routines once. In any case, multiple draw routine calls will almost never occur in practice except under low disk space conditions.

---

[5] This method of background printing should not be confused with the deferred printing or background printing model as provided by the printing architecture. Whether or not the application calls the PrintPage member function in a background task is up to the application. The actual physical printing of the document on the printer always occurs in the background.

The Print Server[6]

Now let's talk about the Print Server.

At the commencement of a print job, the Print Server is notified of the pending job. The Print Server has the responsibility of coordinating all print jobs and to make sure they get printed on the proper printer. The Print Server dispatches a Printer Team to print each print job. If printers are busy, it makes a queue for the job and waits until the printer is free.

The Print Server is not connected in any of the data paths and does not directly receive nor transmit any printing data. It does not interact with the application in any way other than through the Printing Classes. The main service of the Print Server is to dispatch print jobs to printers and to maintain a list of jobs and printers for its clients. The Print Server also allows a user, through a separate user interface, to interrogate the status of any printer or any print job, to redirect print jobs to different printers, or to reprint completed jobs. It also functions as the master switch point for choosing between printers.

Printer Teams[7]

Printer Teams are the third component of the deferred printing model. Printer Teams are actually miniature applications whose job it is to take a print job and render it on a specific device. They are very printer specific and correspond to the classical Print Driver in the classical Macintosh. However, because they are independent teams, they are extensible and very flexible. They can perform extensive printer processing and provide a wide variety of additional services.

Printer Teams are dedicated to particular device types meaning that there must be a different one for each different printer and cannot usually be interchanged. Printer Teams need not be attached to real devices. Devices may be virtual or represent logical devices such as spoolers, remote printers, conversion engines, or simply testing applications.

During the printing process, several Printer Teams may operate simultaneously. The is no restriction on the number and kind of Printer Teams operating at once, except that no two Printer Teams can be attached to the same logical printer.

III. What It Does and How It Does It

Now that we have a general understanding of the overall printing architecture, we will talk about how it functions. This section presents some user scenarios and describes how printing works from the perspective of the application and the user.

---

6 The Print Server is documented in greater detail in a separate section entitled "The Print Server".

7 For more information about the Printer Team and its classes please refer to the separate section entitled "Printer Teams". Printer Teams for special purposes or specific devices can be designed by third party developers as described in that section.

Printing Services

Creating Documents and Assigning Printers

For the purposes of printing, a document is defined as anything that can be printed. Usually it is partitioned into several pieces that correspond to pages, but this is not a rigid rule. A document may be assigned several properties, or settings, such as page dimensions, media preferences, orientation, imageable area, initial printer type, media size and perhaps output requirements (stapling, collation, and so forth); these may correspond to some intended target printer, or not, depending on the application. The application or user has complete flexibility to alter or specify any desired media format, whether it exists or not.

Pages within a document may be assigned diverse properties that are unrelated to the rest of the document, making it possible to have varying page sizes, for example, within a document. Both the document and page properties are stored with the document independently of the printer or printers available. Style sheets or stationery pads may be created as document templates, with predetermined printing characteristics already embedded in them. This, of course, is at the discretion of the application.

Printers, too, have their own set of properties such as page dimensions, media, color and font characteristics. When a document is printed, the document and page properties are matched to the printer in the best possible way subject to the users choice of printer.

Page Setup, Document Format and Metrics

There are two ways to set document properties, first by the user through a human interface dialog and second by the application directly. Either or both methods are acceptable.

When the properties of a document are set or changed by the user, the application first calls the PageSetupDialog on a printable object. This action causes the page setup dialog to appear for the user. Subsequent to, or instead of a page setup session with the user, the application may make additional settings or changes. The application may also use its own dialog or user interface if preferred.

Dynamic Print Properties

Dynamic print properties are those that exist only throughout the life cycle of a TPrintJob. Such conditions may be saved for default settings from job to job, but are normally transient conditions. Examples of these settings are: number of copies, section and page range alternatives, destination (another opportunity to choose a printer), collation, input and output bin selections, and background notification. User authentication or verification can also be performed at this time.

During this phase the application may use the PrintDialog method provided by the system or provide its own.

Printer Selection

The printing architecture allows printers to be chosen by a user and attached to a document. The chosen printer can be set by default for the entire system or it can be directed to a specific document. In fact, different printers can be set for every page of a document.

A printer may be selected as either the current system printer or as the printer for a specific document. The target printer for a document may be chosen at creation time or any time up to its actual printing.

The currently chosen printer is available to a client in the MPrintable class. When a printable object is constructed, a default printer is provided for that object (the current system printer). Subsequently, the default printer may be replaced or changed by the client or the user through the PrinterSelection dialog.

Printers can also be selected on a page by page basis. This service may be necessary where certain pages of a document require certain features of a specific printer. Printing a single document to multiple printers is treated as multiple jobs as far as the printing architecture is concerned, but the client need only instantiate one TPrintJob and doesn't have to worry about which pages go on which printer. Printing to multiple printers can be temporarily disabled when printing draft copies.

Once a printer has been selected for a document, the printer remains attached to that document until the user or client specifically changes it. This minimizes the occurrence of gratuitous document reformatting. If the printer, for which a document has been formatted, no longer exists or is no longer available when the document is printed, the user has several options. The user may specify another printer to use, or let printing default to the current system printer. In either case, the user further has the choice of allowing the document to be reformatted to the new printer, or to be skipped, scaled, tiled or clipped to the new printer metrics. The user need not specify this every time printing occurs, either, since these parameters may be set in a standard list of preferences.

Managing Print Jobs

There is one other user dialog that provides users access to jobs queued up for printing. Once a print job has been dispatched to the Print Server, its status may be requested by the client or user. The JobStatusDialog allows all dispatched jobs to be viewed and altered.

Possible job status properties that may be viewed are: priority, waiting, pending, completed, aborted, failed, or deferred. It is also possible to alter the status of jobs by changing their priority, reordering them in the queue, switching jobs to different printers, or cancelling.

Once a job has finished printing, it is placed in a queue of completed jobs. Until it is removed from this queue, the job may be reprinted at any time by moving it to the queue of any printer. The job-completed queue is flushed at the users discretion or a periodic intervals depending on the availability of system resources such as disk space.

Other Services

Another service provided by printing is job accounting or transaction logging. This service may be augmented by a Printer Team, an application, or remote printer application. Please note that there are no security measures built into the printing architecture and none are planned, but the ability to monitor printer acquisition and usage is allowed.

Multimedia

The buzz word of the day is multimedia. While it is not the intent of this architecture to implement a full multimedia toolkit or application framework, it does provide a multimedia input and output interface integral to the system upon which an application toolkit can be built. By virtue of the developer Printer and Scanner Team mechanisms, third parties are able to couple their own devices into the system using a standard interface, without having to design

one from scratch and without having to integrate it into every new application that comes along.

How does sound fit into the scan and print architectures?

User Interface Samples

These will be included at a later time.

IV. Print Classes and Objects

The Printing Classes were first introduced under the printing architecture section above. If you are not familiar with these classes please refer to that section.

MPrintable

MPrintable is a base class mixin that provides primary printing facilities. Through it an application gains access to all printing facilities. In order to make something printable, it must be mixed with MPrintable or a subclass which inherits from it. It is conceptually quite simplistic but provides a wide variety of services to assist clients in both obtaining information about printing and managing data for printing.

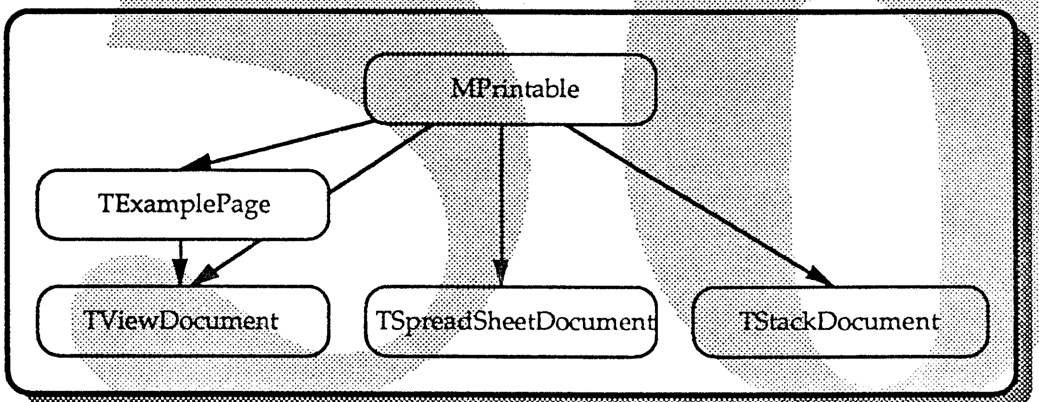


Figure 6. The MPrintable class and some of its descendants.

MPrintable is designed to provide a complete printing framework where very little work needs to be provided by its clients. Or it may be treated as an extensible architecture where many of its services are overridden to obtain exactly the printing model desired. The default model is described first and then in more detail how it may be modified for greater flexibility.

The MPrintable base class may be applied to whole documents, single pages, individual graphics or to anything in between. Any object that descends from MGraphic may be printed as a document in its own right, provided it is mixed with MPrintable. As new components or pages are added to a printable object, they too may be made printable simply by mixing them with an MPrintable and adding them to the superstructure. Because each component has an MPrintable object associated with it, each one may have options that vary from component to component. There may be any number of MPrintables associated with a document, one for the entire document, or one or more for each page.

```
class MPrintable : public MCollectible {
```

```
public:
                MPrintable();
virtual         ~MPrintable();
virtual void    Print(TPrintJob& printJob) = 0;
• • •
};
```

The easiest way to make a printable thing is to create an MGraphic, add MPrintable to it, and use the draw methods of the MGraphic to print. This can be done manually if desired, using the MPrintable from above, or clients may use the MViewIsPrintable class which can be mixed with any TView to make views printable. You don't need to override the Print method because the printing system provides one that should be reasonable enough to use. The class does all the work.

```
class MViewIsPrintable : public MPrintable {
    private:
        TView*          fViewToPrint;
    public:
                        MViewIsPrintable(TView* viewToPrint);
        virtual         ~MViewIsPrintable()

        virtual void    Print(TPrintJob& printJob);
};
```

Default Printing

If an application does not wish to be concerned with MPrintables for each component, Printing will supply one for it based upon one of several default models. The number of models is by no means restricted to these and may in fact comprise mixtures of these models.

General Document Model

A general document is conceptually similar to the traditional idea about documents, namely, that the hardcopy form is a representation of some higher esoteric entity that originates within the mind of the user assisted to some degree by the application tool. Necessarily, documents must be partitioned into page sized fragments before printing can occur. The form and content of each page is set jointly by the application and user. A document consists of an ordered set of pages that are configured to match the available media on a given printer. The ordering is specified by the application but is used by the printer to maintain the proper printing sequence.

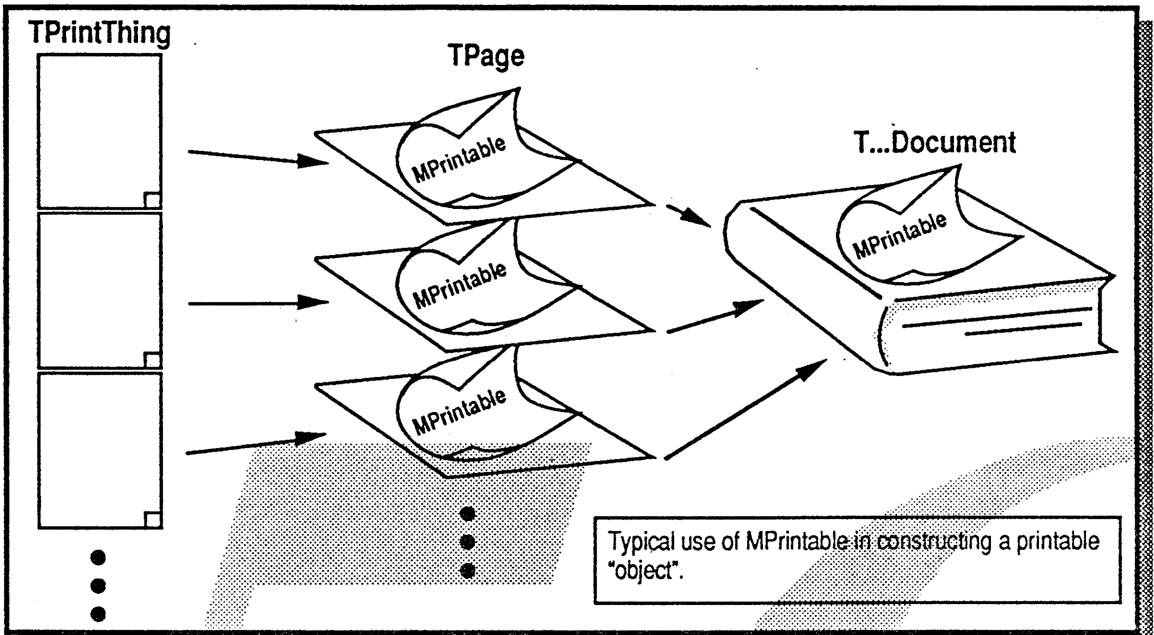


Figure 7. A generic document model. To make a graphic or window printable, it is mixed with MPrintable. These printable objects can then be combined into pages, chapters and sections.

Spreadsheet Model

Another simple printing model is a comprised of a single MGraphic or TView for the entire document. Each page is represented by an arbitrary partition of this MGraphic. The MGraphic covers some arbitrarily large area sub-divided by a grid mesh where the grid size is the size of a page. A spreadsheet or large graphic drawing program might use this model.

This model is necessarily restricted to a constant uniform page size for all pages. Nevertheless, this model will serve a variety of applications very efficiently.

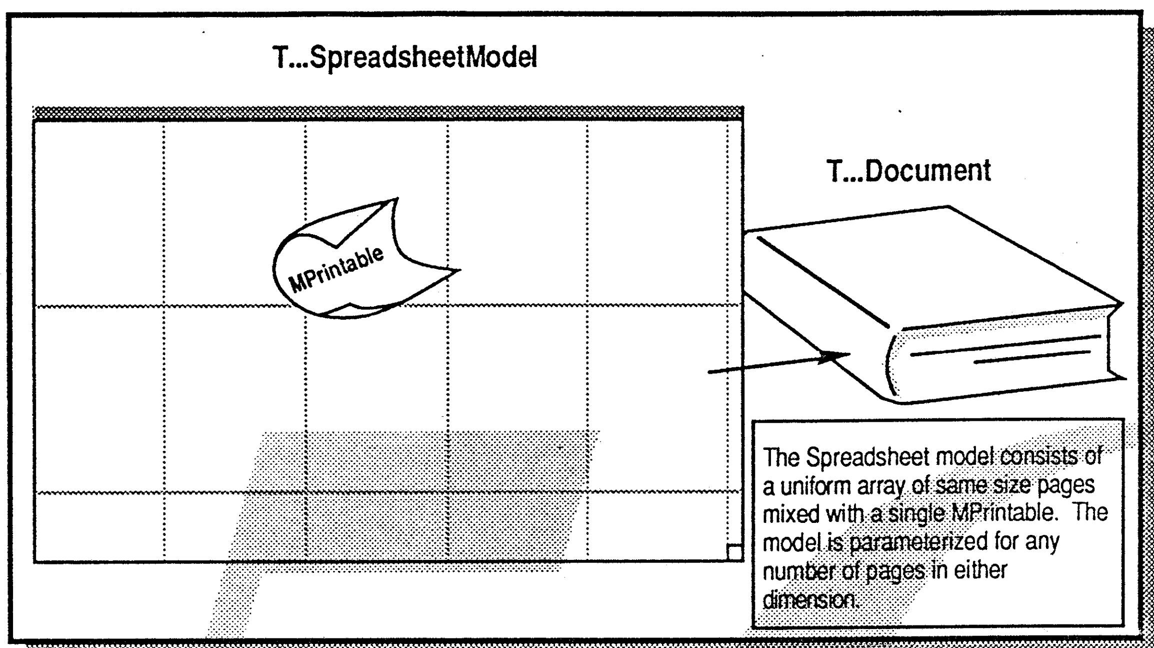


Figure 8. The Spreadsheet Model. Mixing an MPrintable with a large view makes a convenient spreadsheet model for a document.

Stack Model

Another possible choice for default printing is the stack model where each page is represented by a separate MGraphic of the same size. A HyperCard type of application may find this model more suitable than the spreadsheet model. In both models, however, the client is restricted to a uniform page size for all pages.

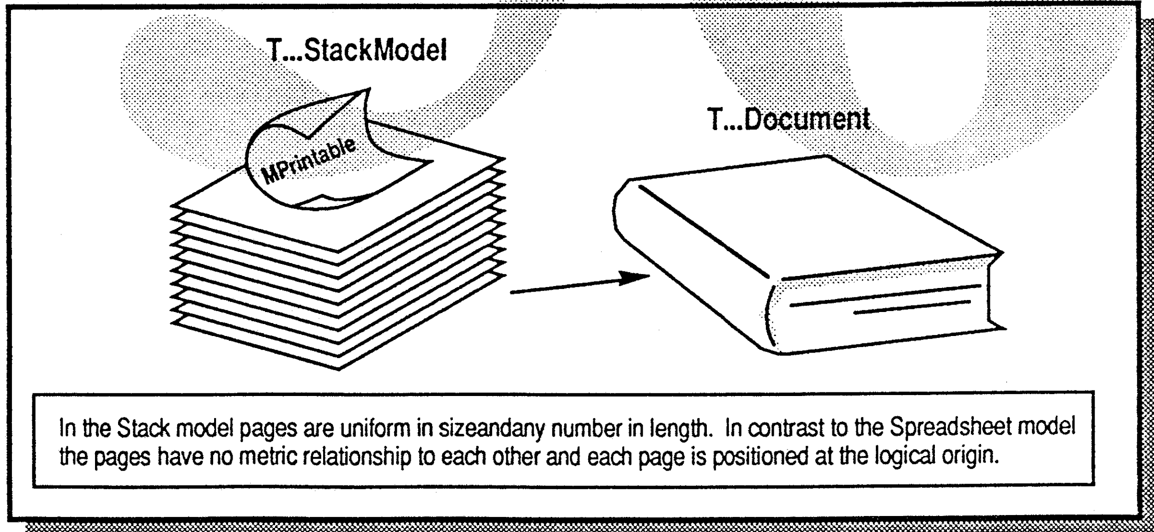


Figure 9. The stack model requires only one MPrintable to represent the entire document.

Pile Model

This model is somewhat similar to the Stack Model except that the pages may have varying sizes and different orientations, and all the pages are stuffed into a sequence.

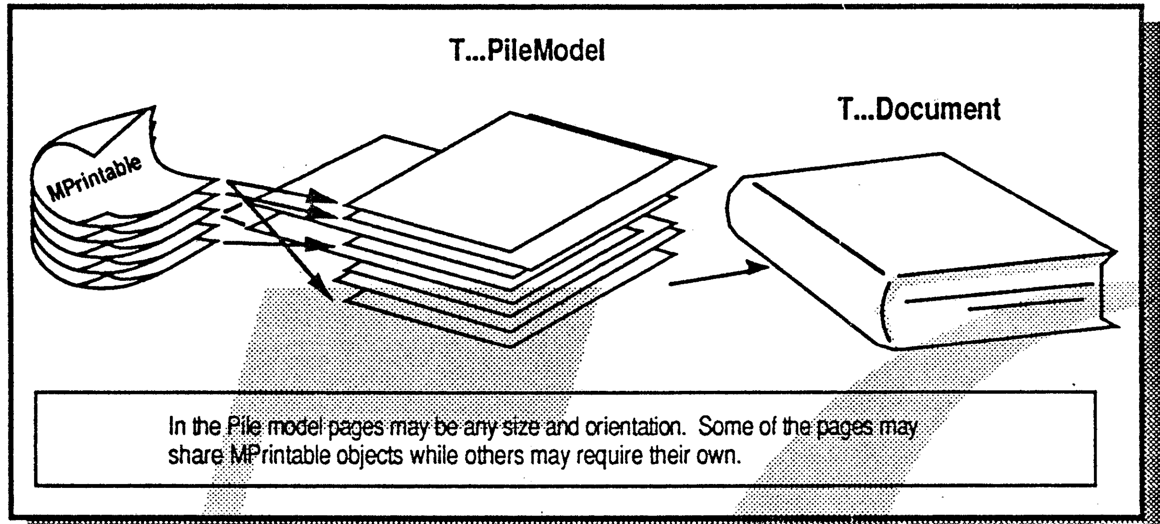


Figure 10. The TPileModel is characterized by variable page types which the client may provide. The pages are maintained by the printing architecture in a standard utility class sequence.

Other Document Models

It is possible for the application to mix these models in a structural prototype of their own design. Completely arbitrary printing models, whose structure has no relationship to the above examples, may be supplied as well. In all models the fundamental framework is the same.

Are there other default models we should provide?

TPrintJob

The other significant class associated with printing is TPrintJob. Its function is to form a print job based on the print parameters for each page. It has many of the same properties of the familiar PrintRecord from classic Macintosh printing. There is really no interaction from the client required for this class except that the application that is doing the printing must instantiate one of these. (We could do this automatically but we may provide other services in the future which dictate that the application do this.)

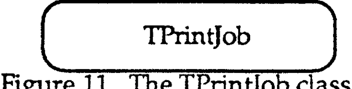


Figure 11. The TPrintJob class.

TPrintJob corresponds to one printing session and exists only for the duration of one job. All the paraphernalia required for a printing session, its state parameters, grafport, and document information, is handled by this class and every print job must have a TPrintJob associated with it.

```
class TPrintJob {
      private:
            TPrintSettings*        fDefaultPrintSettings;
```

```
        TJobSettings*         fDefaultJobSettings;

    public:
        . . .
    // This will eventually be used for printing animation frame sequences as well
        void    PrintPage(MGraphic& graphic, TPageSettings& pageSettings);
        void    PrintPage(MGraphic& graphic);
        . . .
        Get/SetDefaultPageSettings();
        Get/SetDefaultJobSettings();
        . . .
};
```

The fundamental member function within a TPrintJob is PrintPage(). The graphic that is passed to it packages the entire graphical content of the page inside its ".Draw()" call; that is, asking this graphic to draw will result in it drawing everything on the page. Note that a client won't necessarily have to go out of her way to create such a package around the contents of a single page, because if the graphics have been drawn on the screen in a TView, then the TView is already the appropriate packaging, and is what is used when printing a "printable view" (see MViewIsPrintable above).

We can also see in the class definition how TPrintSettings ultimately get communicated to the printing system.

Embedded within the Print Job are a number of parameters that are used to describe the output from a hardcopy device. These parameters are defined by the classes TPrintSettings and TJobSettings. Settings can be assigned arbitrarily on a page by page basis provided there is an MPrintable for each page. Some of the ancestral classes are described later.

```
class TPrintSettings : public MCollectible {
private:
    TPageArea       fPageMetrics;          // Boundary, Printable area, Margins
    TMediaType      fMediaChoice;          // Type of Media
    TDoubleSided    fDoubleSided;          // Double sided printing request options
    TPuntOptions    fSkipTileClipScale;    // What to do if page sizes don't match
    TOrientation    fOrientation;          // Orientation preference
    TSourceTray*    fTraySelection;        // Choice for media source or default
    TDestinationBin* fBinSelection;
    TSet            fSettings;             // Additional client defined settings

public:
                        TPrintSettings();
                        ~TPrintSettings();
. . .
    /* More Stuff */
};
```

Here is what TPrintJob does. Associated with a TPrintJob is a TGrafPort that is like the classic Macintosh GrafPort, but it is hidden from the client. When TPrintJob::PrintPage is passed a graphic, it tells the graphic to draw itself into this private port (possibly multiple times, for instance, to do banding).

The private printing port is a standard Albert TGrafport. However, it is connected to a special "spooling" TGrafDevice called a TRecordingGrafDevice. This TRecordingGrafDevice is associated with a TSegment, the classical "spool file". Rendering calls to the TGrafPort are encoded and written to this TSegment.

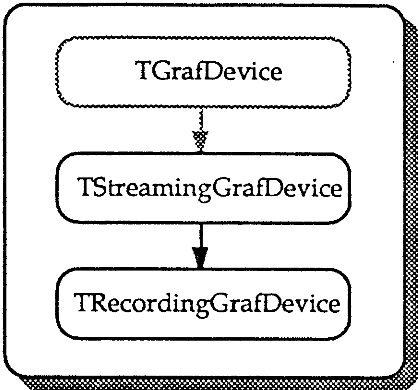


Figure 12. TRecordingGrafDevice. The class used in "spooling and despooling" ("recording and playing back") a print job.

To "despool", the TRecordingGrafDevice is passed to a Printer Team (this is done by streaming/unstreaming the TRecordingGrafDevice object itself. The associated TSegment "spool file" does not get moved or copied in this process). A TRecordingGrafDevice has a member function to play back that which has been recorded and a place to plug in a "real" printer TGrafDevice into which the recording may be played. The Printer Team merely supplies such a "real" TGrafDevice, plugs it into the TRecordingGrafDevice's "output jack", and hits the play button.

TRecordingGrafDevice is designed to deal with simultaneous recording and playback (spooling and despooling). There are also facilities for working with a small amount of disk space where a finite sized spool file is reused over and over again in a process like this: "Spool a little, despool it, rewind the spool file and spool a little more, despool it all, and so forth."

Support Classes

The following classes make up the parts of TPrintSettings & TJobSettings that characterize a document for printing. These settings may represent a real printer, but don't necessarily have to.

Printers in the pink world are not necessarily dependent on paper as a primary media, and therefore, the word media has been chosen to refer to all types of printer output including paper.

Media Classes

Media is the term used to specify what a document page will be printed on. Media is characterized by a size metric, an ID, and a human readable name. The name is localizable, that is, it may be altered for local languages where appropriate. The identification token serves to distinguish every media type from every other, even if the name has been localized. Real printers have at least one, and perhaps many, media types and one class associated with each type. Printers are allowed to have any number of media types. (See Printer classes below.)

TMediaType

Matching media types between different printers is done first by ID and then by size.

TMedia may be subclassed to provide more detailed specifications to the document, such as film type, gamma index, tone reproduction curve, and so forth.

Trays and Bins

Closely associated with media type is the input tray, or source of the media; and the output bin, or page destination. Notice that the word "tray" is used to designate input sources and the word "bin" to designate output destinations. A tray or bin may be logical entities, meaning that they do not have to represent actual hardware on the printer. For example, two types or two sizes of paper may actually originate from the same physical tray, but logically they are treated as two trays.

The number of trays or bins attached to any printer is completely arbitrary and is determined by a specific Printer Team. Printer Teams can be dynamically altered to change the number and type of trays or bins to accommodate new devices or field upgrades. The mapping of trays to bins for a particular job is determined first by the capabilities and services provided by a specific printer being used for output, and second, by the requirements of the document being printed. The user may, of course, also dictate specific trays and bins.

Trays and bins act as containers for printing media and hence are a subclass of the media class. Some instances of the container class may require that the media section be replaceable or interchangeable and this is entirely appropriate.

The next logical extension is to allow for arbitrary collections of media, trays and bins and for these collections to represent the collections of trays and bins on an actual printer. Collections can be dynamically altered to allow for users to change trays, bins and media. Synchronization between the collections and a physical printer will definitely be a problem so media content may sometimes be unspecified and the user will be responsible for maintaining synchronization.

Trays have an imageable area associated with them which represents the physical limitations of the printer in printing all the way to the edges of the media. The imageable area is enclosed by the total media area. It may be less than or equal to the media size, but it can not be any larger.

Trays also have an orientation that represents how the image is printed in relation to direction of media travel in the printer. Note that orientation of the media is not, repeat NOT, the orientation of the image on the page, since this definition is meaningless in some contexts. Media orientation is defined with respect to the direction of travel of the media through the printer and how it lands in the output bin. Image orientation is superimposed over the possible media orientations to generate eight possible configurations. This allows the user to see exactly how the image will appear with respect to the media as it is placed in the printer input tray which is really what she wants to know anyway.

Iconically, orientation may be represented as shown in the following figure.

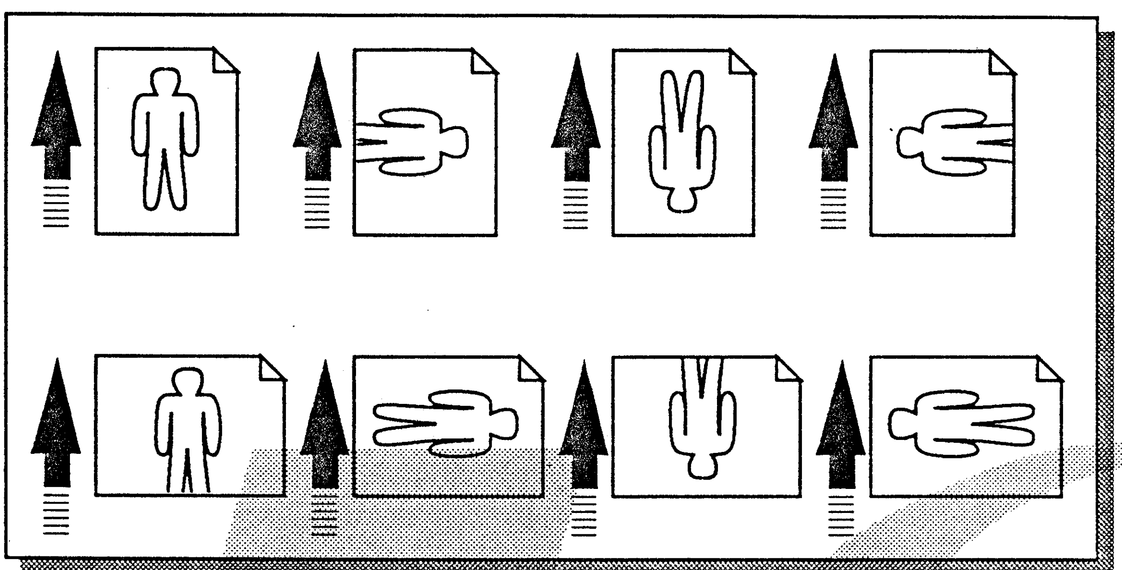


Figure 13. Possible media orientations with respect to paper path through a printing device.

The possible orientations are represented as an enumeration type.

```
typedef enum MediaOrientation { NarrowUp, NarrowRight, NarrowDown, NarrowLeft, WideUp,
        WideRight, WideDown, WideLeft };
```

Note that the only real distinction between the Narrow and Wide orientations is in the width of the media versus the length. On some printers this distinction may be totally lost or be completely interchangeable where width equals length.

Stacking order describes how pages are stacked in the output bin with respect to previously printed pages. Possible alternatives are face-in and face-out.

```
typedef enum StackingOrder { Faceless, FaceIn, FaceOut };
```

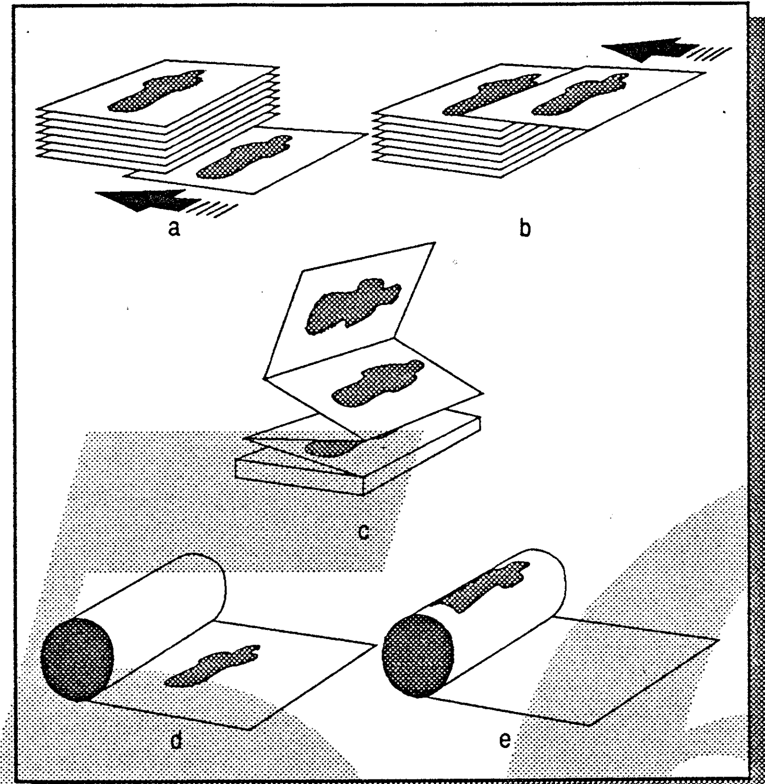


Figure 14. Stacking order defined. (a) Face-In (b) Face-Out
(c) Faceless (d) Face-In (e) Face-Out

As seen in the figure above, face-in and face-out have nothing to do with the page relationship to the bottom or top of the output stack, but simply whether the printed image on the output, faces into the previous page or out of it. If the output stack were upside down or sideways it would have no effect on whether the pages were face-in or face-out. With fan-fold media, stacking order is meaningless, and with roll media, although the meaning is clear, the relevancy may be negligible. In the latter case the designation could be termed Faceless.

In the case of duplex printing (printing on both sides), the very same nomenclature applies except that fanfold and roll media are perhaps less representative of the real world. The following figure shows the representation of duplex printing, but in this case, the term face-in applies to the most recently printed side of the current page as it faces into the earliest printed side of the previous page. Face-out is just the opposite, the earliest printed side of the current page faces against the last printed side of the previous page.

When duplex or double sided printing is available, it is important to maintain the page order between the two opposing faces. A duplex tray therefore has a sequence associated with each page of media it delivers. This must specify how the second face must be printed in relationship to the first face. For example, it could be immediately following the first face or after all first faces have been printed.

TDuplexOrdering

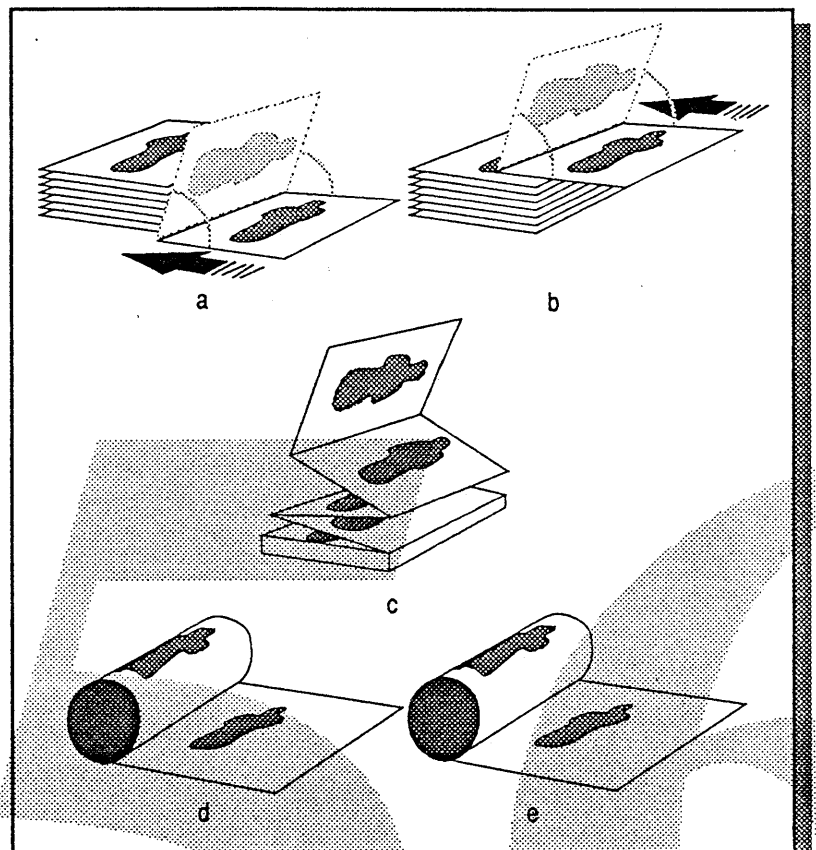


Figure 15. Duplex printing with various kinds of media. (a) Face-In (b) Face-Out (c) Faceless (d) Face-In (e) Face-Out.

In this diagram we are tacitly assuming that the side of the page facing up is the most recently printed side in order for our definition to hold. Otherwise, the labels for (a) and (b) should be swapped. Notice that even if both sides are physically printed at the same time, the definition still applies logically; that is, the paradigm is mapped back to the data stream where the side of the media printed with the first page of received data is considered to be the first page physically printed. Note, too, that (d) and (e) are almost identical except for the order in which images are placed on the media. As far as usage is concerned, they may be considered completely equivalent.

When a document is created the media size and format for each page are specified either by the application or the user. At print time each page of the document is matched with the appropriate input tray and the proper destination bin. The criterion for matching document pages to input trays is media size, type, and whether or not manual intervention is required. For bins the criterion for selection is collation, finishing needs such as stapling or binding, or whether a specific address is required such as a mailbox. The ultimate selection can of course be manually overridden.

Printer Classes

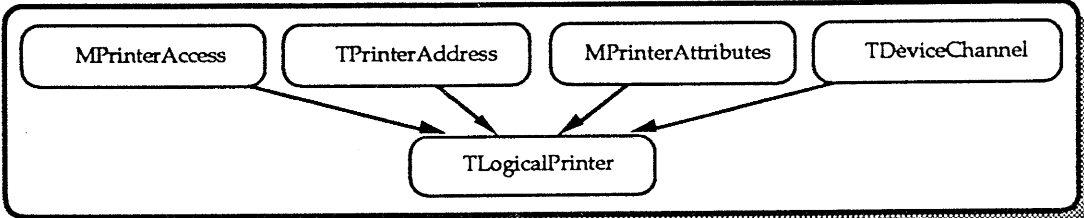


Figure 16. The Printer Class which represents the chosen printer for a document.

The class that represents a real printer type is a generalization of many printer characteristics. Printers are a conglomeration of several other classes; a GrafDevice, media, and Tray and BinOptions. A printer choice does not have to represent an actual printer. It may be a logical printer or represent a printer not physically connected to the users machine. Printer classes are dynamically allocated based on the currently selected printer or currently opened document (which has a selected printer).

Printer Addresses

Every printer has an address associated with it that defines its name and location. This material is necessarily only a hint to the actual location since printers are not (yet) able to sense their own location. It is intended primarily for use as a means of identification for the user but may be meaningless on some printers especially those unable to store their own location within themselves.

```
class TPrinterAddress {
public:
        TName& GivenName(TName &); // Provides name of printer if it exists
        TName& ProductName(TName&);// The Apple Product Name
        long PrinterID();        // Unique for every printer type in the world
        TName& Location(TName&);// Arbitrary text describing physical location
};
```

A printer address consists of several descriptive text strings perhaps arbitrary or optional. The ProductName should correspond to the product name used in manufacture. Note that the printer address does not necessarily correspond to an electronic address or physical IO port since these are embedded within the Printer Team itself.

Printer Attributes

In addition to addresses, printers also have attributes.

```
class MPrinterAttributes : public TPrinterAddress, public TTrayOptions,
public TBinOptions {
protected:
        . . .
        long Version();    // Returns the version number of this class
        TPrinterAddress& Address(TPrinterAddress& address);
        Point& Resolution(Point& DeviceResolution); // Pixels/in2
        TDuplexity DoubleSidedPrinting();  // one or two sided printing
        Scan FieldofScan();          // Direction, Raster, vector…
        Cardinal FieldDepth(); // Bits per pixel per color
        GraphicModel ImagingModel(); // QuickDraw, Albert, PostScript, raw…
        ColorModel ColorModel(); // noColor, monochrome, spotColor, full Color…
        Color Background(); // white vs. black… Default background color
        ColorTable ColorMatching(); // Color gamut of device
```

```
};
```

The attributes class is a mixin that provides device information about the attributes of a specific printer. Normally the application does not access these attributes since all information necessary to format a document is either accounted for in printing or is provided by other means. All of these properties are mentioned here for completeness.

Device Access

Device Access refers to a users privilege to acquire and use a printer for printing. In most cases users are always given access, but in a controlled environment where monitoring or charge accounting is required, access may be logged through this class.

Access to printing devices is controlled by MAccess, a mixin provided to coordinate access to a logical printer. Default access is always initially wide open to everyone. Applications, Printer Teams, or other utilities may monitor or restrict access to certain devices by implementing an access base with the following class. The access class is not intended to provide the ultimate security in access control but is meant to provide a hook for applications to add on flexible extensibility to the printing environment.

```
class MAccess {
public:
        . . .
        virtual Boolean Access(Name, Password, IDNumber);
        . . .
};
```

Logical Printer

```
class TLogicalPrinter : public MPrinterAttributes  {
        private:
                TID     fIdentity;
        . . .
};
```

V. Examples

Setting up a Printable MGraphic

To accomplish printing, the client creates a MGraphic from which she desires to obtain hard copy. This may be an ordinary window view or some other combination of MGraphics. One possible example is a view created from a TView class and an MViewIsPrintable as follows.

```
class TPictureView : public TView, public MViewIsPrintable {
        public:
                TPictureView() : TView(TGPoint(1400, 1400), TGPoint(0.0, 0.0)),
                                MViewIsPrintable(this) {}
                virtual ~TPictureView() {}
                virtual void DrawSelf(TGrafPort *port);
};
```

In this example, the only thing the application has to provide is the DrawSelf method which is just the same stuff that the view draws. The same drawing routine that draws to the view can be used to print.

Paging

This example does not take into account the paging model for the view. Normally, the view may be associated with a single page, part of a page or many pages depending upon the model the client has in mind. In this example, paging is not taken into account, but the view could represent a single page, or any number of pages with the same print settings. If the view is not the same size as the output page on the physical printer, printing will still be performed, however the page to document wysiwyg correspondence cannot be guaranteed. The client has three choices for the output format when this occurs, the final output can be scaled, tiled, or clipped. The clients choice can be registered by setting TPuntOptions in TPrintSettings by calling SetPuntChoice with an argument of kPrintTiled, kPrintClipped or kPrintScaled. The default choice is kPrintClipped.

Scaling forces the view to be scaled to the output size of the page, while tiling causes the pages to be partitioned into a uniform grid the size of the printable area, similar to MacDraw. Clipping is self explanatory.

Programmers, take note. The punt options are not the mechanism for controlling paging and document layout. They are not to be confused with the paging model. Its sole purpose is to resolve conflicts between view and physical page size differences.

Printing

To do the actual printing the client does something like:

```
TPictureView*    gPictureView = new TPictureView;
TPrintJob        printJob;
TPrintSettings* settings = printJob.GetSettings();
settings->SetPuntChoice(kPrintTiled);
gPictureView->Print(printJob);
```

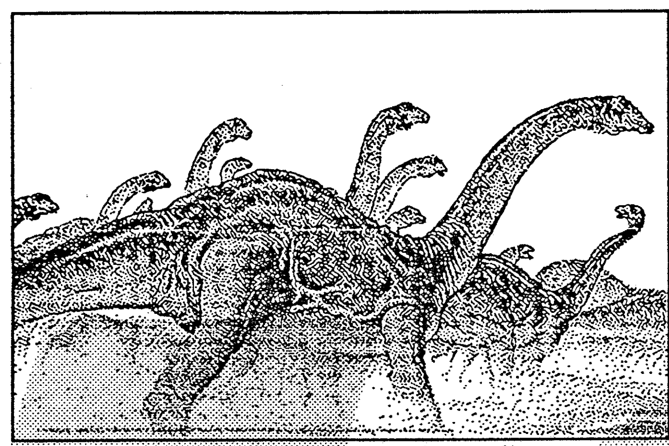
56

The Outer Limits

The Print Server

56

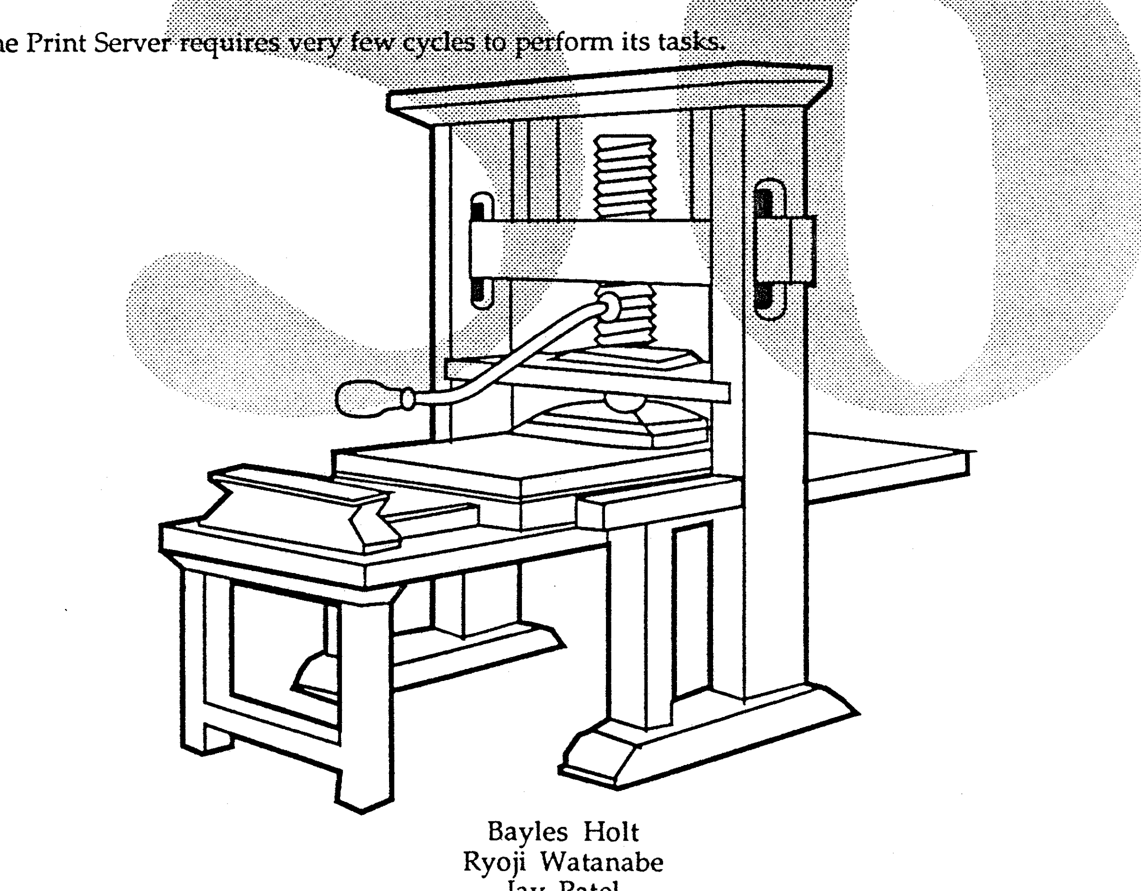
The Print Server



Noted for its size and speed!

The Print Server is responsible for managing print services which, among other things, include providing access to printers and Printer Teams, and dispatching print jobs to printers. It manages the queue of print jobs associated with each printer and allows clients to modify job order and priority in the queues. The Print Server also perfoms some name management associated with printing as well.

The Print Server requires very few cycles to perform its tasks.



Bayles Holt
Ryoji Watanabe
Jay Patel
Mahi deSilva

56

Table of Contents

The Print Server

The Print Server....ii
Table of Contents....iii
Table of Figures....iv
I. Introduction....1
II. Architecture and Overview....2
Printer, Printer Teams, and Print Job Lists....3
Print Job Queue....3
Print Job Dispatch....4
III. How It Works....5
Printing....5
Other Services....6
IV. Print Server Classes....6
Class Overview....7
Class Member Functions....7
V. Examples....7

56

56

Table of Figures

Figure 1. The Primary Print Server class..1
Figure 2. The Print Server client class...1
Figure 3. The Three Queues maintained by the Print Server...2
Figure 4. The Print Server provides a list of Printers and Printer Teams 3
Figure 5. The Print Server provides a list of print jobs...4
Figure 6. The Print Server dispatches new print jobs to individual Printer Teams 5

56

I. Introduction

The contents of this section are best understood if the overall printing architecture is already understood. Please refer to the "Printing and Imaging Architecture Overview" section.

We begin by referring directly to the Print Server class, a subclass of MServer, and to its client MClient.

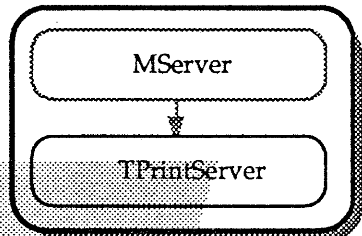


Figure 1. The Primary Print Server class. This class exists only in the Print Server itself.

The Print Server is a normal server, inheriting its main features from the MServer class. Clients do not normally have anything to do with this class, but instead communicate with the server through the TPrintServerClient class or MPrintable classes.

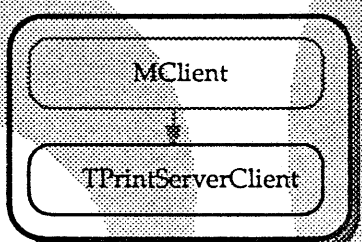


Figure 2. The Print Server client class. An instance of this class is created in the client wishing Print Server attention.

Even then, most applications will not directly instantiate a TPrintServerClient class since this is handled automatically by printing. However, the structure and operation will be described here so that clients can appreciate how the Print Server functions.

The Print Server is relatively transparent to most applications. Most of its services are provided through upper level classes or from other services in the system, however, it is possible to communiciate directly with the server if need be.

The Print Server performs five basic functions.

- Provides a hierarchical list of available printers accessible to the system.
- Provides a list of Printer Teams that are dedicated to specific types of printers.
- Provides a client alterable list of print jobs for each printer.
- Queues print jobs for dispatch to each Printer Team (and eventually to specific printers).

- Displays the print job queues for the user to access. This is the only foreground activity peformed by the Print Server.

The sequence and priority of the items in the print job queue can be arbitrarily altered or reordered. Some items can be frozen or made to require a password and authentication before alterations are allowed. Clients making such restriction on queued items must be authenticated before regaining access. Restricted items can also be prevented from being processed until authentication is provided.

The queues created by the Print Server inherit from the basic deque and collectible classes.

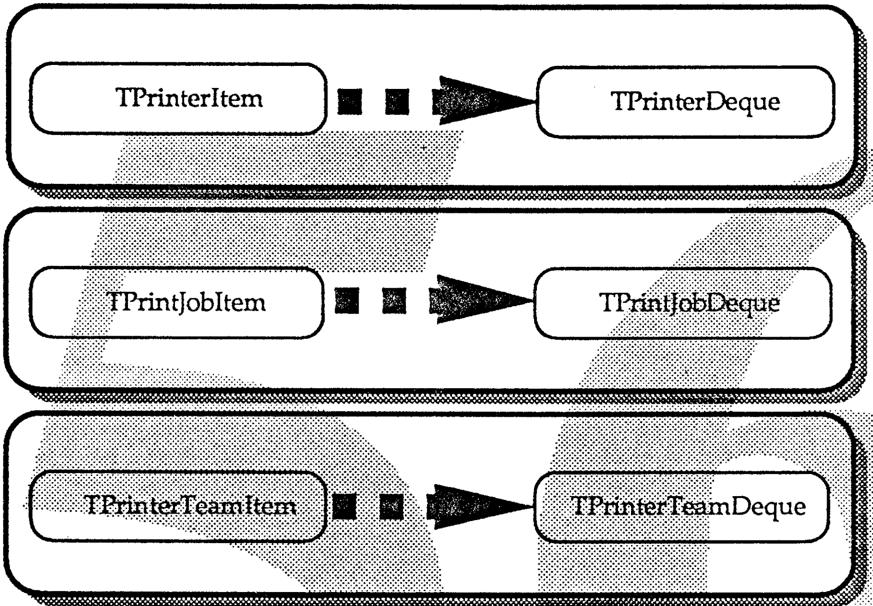


Figure 3. The Three Queues maintained by the Print Server, Printers, Jobs, and Printer Teams.

There are three main queues, one for printers in the system, one for print jobs requested by the applications and one for Printer Teams which drive the printers.

II. Architecture and Overview.

The Print Server is a stand alone team of tasks that is initiated by the system at startup time or by any application that accesses printing services. When the Print Server is first invoked, it goes through a flurry of activity checking for consistency among all the printers, Printer Teams, and queued print jobs, after which it settles down to leisurely replys to requests for services.

If for some reason, such as a power failure or emergency shut-down, print jobs are left unprocessed, these are found by the Print Server at system startup and returned to the appropriate queue for processing. If the job cannot be processed for any reason, the user is notified and the job deleted.

The services provided by the Print Server, as outlined above, may be grouped into three categories:

- supplying lists of printers, Printer Teams, and print jobs.
- providing a user interface through which clients may alter job queues.
- dispatching print jobs to idle printers.

We'll examine each of these categories a little more in the following sections:

Printer, Printer Teams, and Print Job Lists

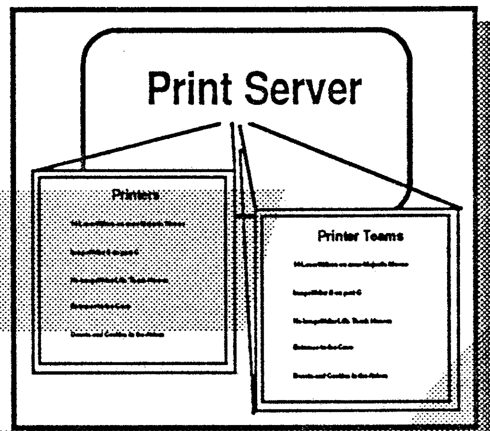


Figure 4. The Print Server provides a list of Printers and Printer Teams available in the system.

The Print Server provides lists of printers, Printer Teams, and print queues to client callers. The interface for doing these operations is described in later sections. The lists of printers and Printer Teams are not directly alterable and are provided only as a service for perusal. Items in the print queues, however, may be altered, reordered, deferred for later printing, or swapped to different printers. New printers, Teams or jobs may be added to the lists by sending the appropriate requests to the server. Normally a new print job is automatically queued whenever the application prints, but applications with special requirements may add jobs to the list directly, provided that the requests are not bogus. The mechanism for doing this is not complex and will be described later.

Print Job Queue

The Print Job queue is shown iconically in the next diagram.

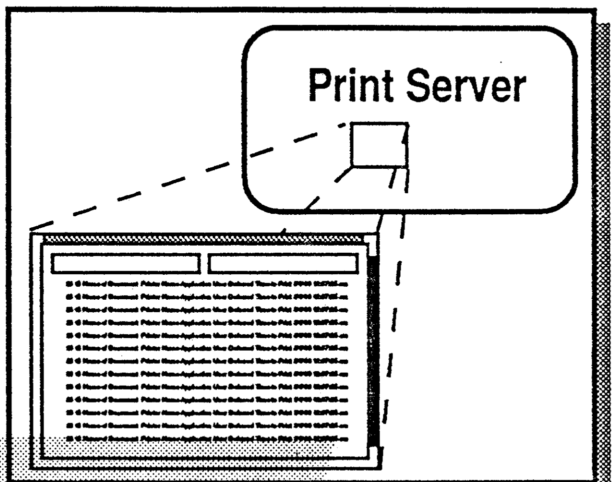


Figure 5. The Print Server provides a list of print jobs assigned to each printer.

The Print Server provides a user interface for presenting the system's printers and all the jobs currently on queue for each one. The interface indicates the printing status and allows jobs in the queues to be aborted, removed, reordered, switched to different queues, duplicated, deferred for later printing, or reprinted. There is one queue for each printer and another queue for recently finished jobs.

Users move jobs around in the queues by selecting and dragging them from one queue to another, even across window boundaries.

Print Job Dispatch

The most significant job performed by the Print Server is dispatching jobs requested by applications to the appropriate printers for printing. The following diagram indicates how this is done.

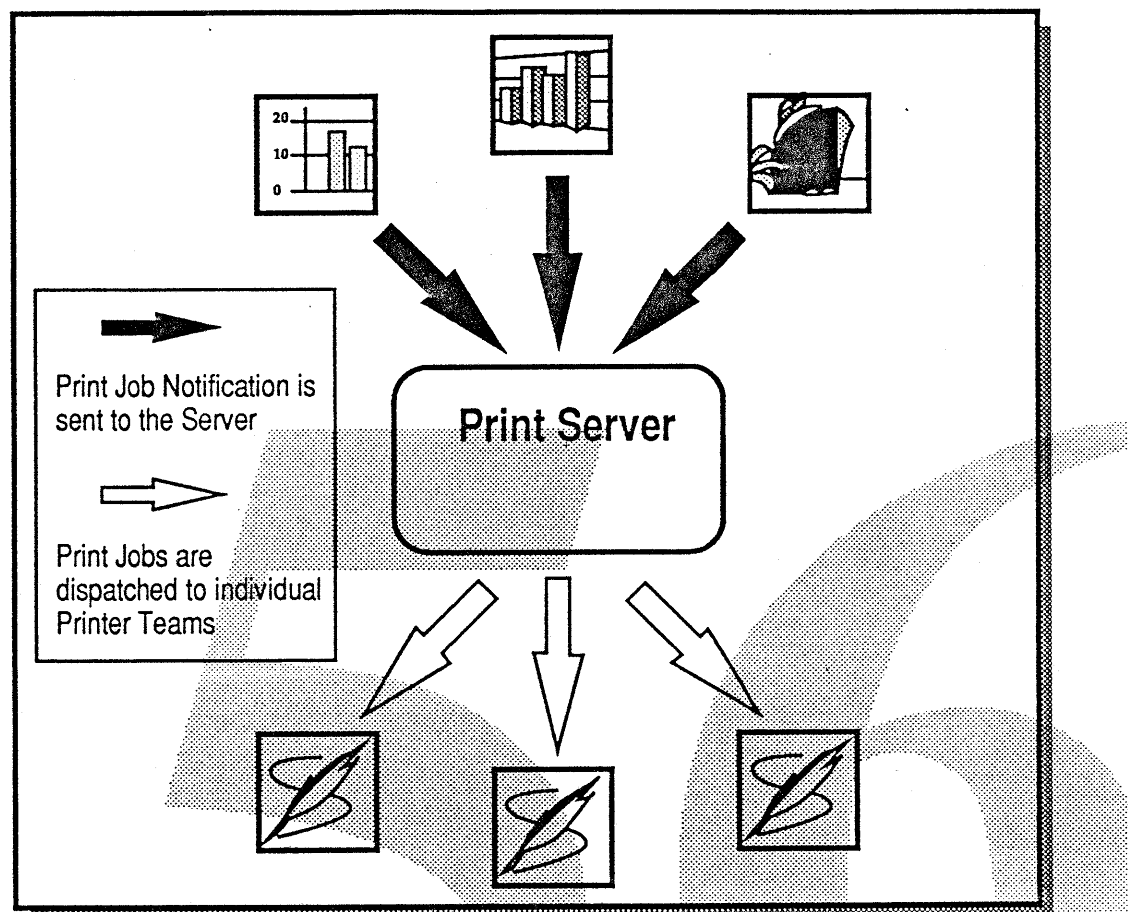


Figure 6. The Print Server dispatches new print jobs to individual Printer Teams.

When an application prints, notification is automatically routed to the Print Server without the application being aware that anything but printing is going on. The Print Server then automatically dispatches the request to a printer for processing. Every printer is controlled by a specific team called a Printer Team that is dedicated to that particular type of device. The particular Printer Team invoked is the one chosen by the user and is typically stored with the document when it is printed.

III. How It Works.

Printing

The primary purpose of the Print Server is to dispatch print jobs to the correct printer. It does this by invoking Printer Teams, giving them a scheduled print job and assigning them to a specific printer. It performs coordination between all printers, printing jobs, and printer queues. It also functions as the master switcher for choosing between printers. It is not connected in any of the data paths as it does not directly receive nor transmit any printing data. It does not interact with the application in any way other than through the printing interface.

To see how a job gets queued through the Print Server, we will run through a sample scenario beginning with the user in the application.

The user initiates a printing job in the application (by clicking OK in a Print dialog). The application begins the printing process by calling the Print member function in the printable class object. During printing, notification is automatically sent to the Print Server that a job is ready for printer queueing. The Print Server takes the job and dispatches it to the appropriate printer, invoking a Printer Team if one does not currently exist. The job may either be processed immediately or queued for later printing, depending on the nature of the job, the availability of the required printer, how many pages, the number of copies, the availability of system resources, and the capabilities of the printer. If the queues are full and all printers are busy, the printing application thread may be blocked until a printer becomes available or a queue slot is opened up. This normally never occurs unless disk space is used up.

Other Services

Applications may query the Print Server for information about printers, Printer Teams or jobs. Queries may be made asynchronously by anyone. It is also possible to modify the job queues for any of the printers in the list but clients had better know what they are doing. (And don't think I am going to tell you.) Each of the print jobs in the queue has a status associated with it which an application can use to keep a user apprised of its progress. The user, aside from the application, may also directly access the Print Server for this status as well.

IV. Print Server Classes.

TJobToPrint

A TJobToPrint represents a print job needing print service. It is created by a TPrintJob in the Printing architecture framework by the application. This thing contains everything that the Print Server and Printer Team need to know about printing the job. The TJobToPrint is passed to the Print Server when it is ready.

Applications that do default printing normally do not need to access any of the Print Server classes unless services other than printing are needed. Printing services are performed automatically by the Printer Classes in the application. In the event that an application needs to get to the Print Server anyway, there is one main client class provided to access any of the Print Server services.

Since the operation of the Print Server is totally client driven and all its services are reached by member function calls, the best way to describe its operation is to describe its member functions.

```
class TPrintServerClient: public MClient {
      • • •
public:
                        TPrintServerClient();
      virtual void      Hello();
      // The next member function will undoubtedly change
      virtual void      PrintAJob(TJobToPrint,...);
      virtual void      PrintDialog(...);
      virtual void      SetupDialog(...);
      virtual void      MonitorDialog();
      virtual void      PrintAbort(...);
      virtual void      PrintAbort(...);
      virtual void      AbortAll();
      virtual void      PrintReschedule(TJobToPrint,...);
      virtual void*     GetSystemPrinter(...);
      virtual void*     GetDefaultPrinter(...);
```

```
	. . .
};
```

Class Overview

There are two ways to activate the Print Server, either print or instantiate a TPrintServerClient class object and call any of its member functions. If it is not already activated, the Print Server will be created, initialized and started. The request will then be communicated to the server for processing. If the Print Server is being invoked for the first time there will be an initial delay while the Print Server goes through its initialization process. Under some conditions this could take several seconds.

Class Member Functions

TPrintServerClient::Hello insures that the Print Server is awake and active. This method normally does not have to be called by anyone. It is provided only as a means for invoking the Print Server when no other services are required.

TPrintServerClient::PrintRequest creates a print job for a particular printer. The TJobToDo contains all the necessary information to print the job.

TPrintServerClient::PrintDialog raises the user dialog for the given printer in question. (This may want to be a process of handing a dialog back to the application for it to raise.)

TPrintServerClient::SetupDialog raises a page setup dialog for the given printer. (This is liable to change drastically over time.)

TPrintServerClient::MonitorDialog raises a printer and job queue dialog for all printers. Any user can modify the content order of this dialog. (This is liable to change drastically over time.)

TPrintServerClient::PrintAbort aborts the current print job on the specified printer or the specified print job on the specified printer.

TPrintServerClient::AbortAll aborts all printing.

TPrintServerClient::PrintReschedule takes the given job on the given printer and places it in the queue of a new printer at a new position.

TPrintServerClient::GetSystemPrinter returns the chosen printer specified by the user has specified for general system printing.

TPrintServerClient::GetDefaultPrinter takes the given job on the given printer and places it in the queue of newprinter at new position. If new is zero it appends to the end of the queue.

V. Examples

To activate the server from an application during normal printing, do nothing. Activation will occur automatically.

Other services may be obtained with code something like the following:

```
TPrintServerClient TellTheServer;

TellTheServer.Hello();  // Make sure she's running
. . .
```

```
// Create or TJobToPrint (from a TPrintJob, maybe) and pass it to the server
TellTheServer.PrintRequest(TPrintJob);
/*
   Actually an empty TJobToPrint or any subclass
   can be passed as a job to the Print Server.
   All that is required is that the Printer Team for the
   printer in question be able to understand and decipher it.
*/

TellTheServer.MonitorDialog();    // Show the user what we just did

TellTheServer.AbortAll();     // Then kill it just for spite
• • •
```

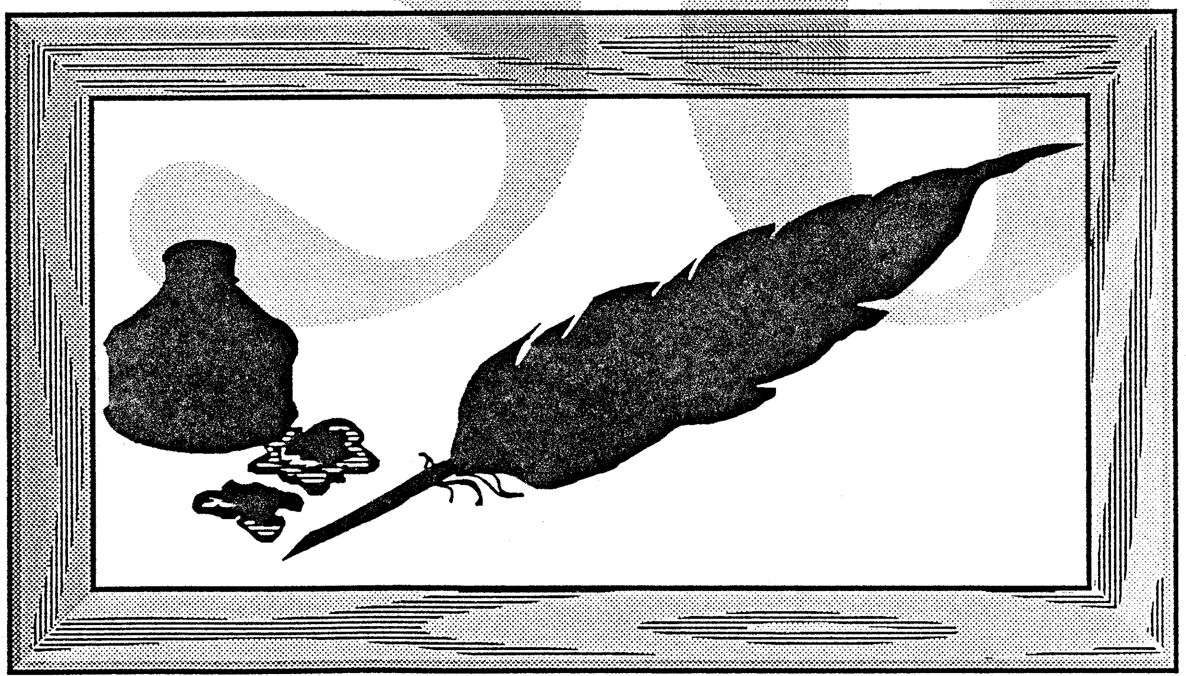
The Outer Limits

Printer Teams

56

Printer Teams

Printer Teams drive printers. This document describes how a Printer Team works, how to write one for a specific printer and how to talk to one.



Bayles Holt
Ryoji Watanabe
Jay Patel
Mahi deSilva

56

Table of Contents

Printer Teams

Printer Teams....................ii
Table of Contents....................iii
Table of Figures....................iv
I. Introduction....................1
II. Architecture and Overview....................1
"What You See Is What You Get"....................2
Single Printer Team Interface....................2
User Dialogs....................2
Printer Selection....................2
III. How it Works....................2
IV. Classes....................4
TDeviceChannel....................4
TPrinterTeam....................6
TRecordingGrafDevice....................6
V. Examples....................7

56

56

Table of Figures

Figure 1. Printer Teams 1
Figure 2. Several Printer Teams can all be operating at once 3
Figure 3. Several Printer Teams operating at once 4
Figure 4. The Printer Team Class 4
Figure 5. Printer Teams drive printer devices 5
Figure 6. TRecordingGrafDevice 6

56

I. Introduction

Printer Teams are comprised of two components, a mini-application that drives printers and a shared library resource that provides user dialogs for that printer. When we say "Printer Team" we usually mean both, or it is obvious from the context which component we are referring to. If there are any ambiguities we will try to explicitly say which piece we mean.

The "Team" portion is an autonomous team that executes in its own address space and controls a specific printer. The shared library portion supports dialogs and other user interface services, which are executed in the address space of the application.

In colloquial terms a Printer Team is like a classical print driver, but it has the added characteristic of being a completely autonomous application, made up of any number of individual tasks (hence a team) and it operates totally in the background. It is actually more analogous to the ubiquitous "Print Monitor" renown in the classical Macintosh, but in the Printer Team case there is a unique team for every printer in the world, and multiple instances of the Team can exist at the same time. Also, unlike the Print Monitor, Printer Teams do not explicitly control how print jobs are passed to it. All jobs that it handles are received one at a time from the Print Server.

A Printer Team is dedicated exclusively to one specific type of printer. Whenever a new type of printer is obtained, a new Printer Team must be written to drive it. Printers without Printer Teams associated with them are not visible to documents, and documents cannot be printed on them using the standard printing architecture.

It is possible to customize existing Printer Teams to add new features or parameters to already existing printers. Individual printers may gain alternate input paper trays, optional color modes, or selectable document finishing characteristics. A Printer Team has personality traits that allow it to be configured for a particular printer with a given set of options or it may be customized dynamically by a user or application to fit a given set of printer attributes.

II. Architecture and Overview.

The Printer Team architecture is illustrated in the following diagram. The Printer Team is a component of the overall Printing and Imaging architecture outlined in a previous section. In this section we will concentrate on the Printer Team itself.

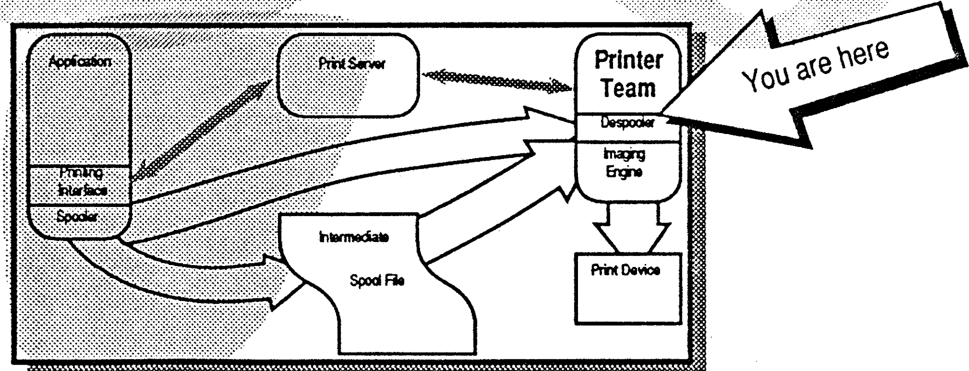


Figure 1. Printer Teams drive printers. From the overall architecture, we focus on the Printer Team portion.

Printer Teams are like classical printer drivers. Each one can drive only one specific type of printer. Because they are autonomous teams, however, any number of them can be operating at once, even for the same type of printer. A Printer Team of that type is instantiated for each physical printer being used.

A Printer Team receives its all its commands and instructions from the Print Server. When it is activated, it receives a single print job, which is usually contained in a Spool File created earlier by a document application. It processes the job, prints it on the device assigned to it and goes back to sleep.

"What You See Is What You Get"

A Printer Team is responsible for reproducing as closely as possible what was intended by the document application. This specifically means "WYSIWYG" fonts, graphics and color matching, within the capabilities of the printer being used of course. Each hardware device has a specified font set, calibrated color gamut, and known reproduction characteristics which are characterized in the Printer Team. These parameters enable the Printer Team to transform any text string or graphic into the best reproduction available on the printer, matched in color, position, size and appearance.

In addition, global fidelity is also maintained.

Single Printer Team Interface

The Printer Team can be thought of as a black box with a standard input and output associated with it. The input object is a TRecordingGrafDevice (documented in a previous section) received from the Print Server. Output is a TDeviceChannel, described shortly.

User Dialogs

Printer Teams provide other functions as well as printing. When an application or document raises a Page Setup or Print dialog, those requests are dispatched through the Print Server to a Printer Team. The Printer Team acts like a shared library in the documents address space, providing the necessary resources to launch the requested dialog. The dialogs may be invoked readily whether or not there is active printing going on at the time, and are completely independent of any other part of the Printer Team.

There will be a bunch more of this stuff to come.

Printer Selection

The Printer Team also provides a selection framework, a way for users and applications to choose a printer. These are implemented as shared libraries and are used exclusively by the Print Server. The Printer Team selection package includes such things as personal icon, selection parameters, access protocol, and hardware connection.

There will be a bunch more stuff on this to be announced.

III. How it Works

To understand how a Printer Team functions, we will present a simple printing scenario and describe what the Printer Team does within this process.

Suppose a user wishes to create a new document in some application. First she probably has in mind some particular type of printer for which the final printout is being targeted. The chosen printer is attached to the document by choosing PageSetup in the context of the current system printer or selecting a printer as a template for the document. The user then builds the document.

When the document is completed, the user asks for the document to be printed. This is done in some iconic manner through a Print dialog, for example, which accesses the intended Printer Team, or by dragging and dropping the document on the intended printer icon. The print parameters may be modified directly by the application or through the Print dialog if the user invoked it.

When the user has made her printing choice selections the document is spooled to disk and the Print Server is notified of a completed print job. The Print Server catalogs the request, attaches it to the appropriate queue and activates the proper Printer Team to process the job. The Printer Team receives the streamed print job through a TRecordingGrafDevice and converts it to the printed page. Color matching, WYSIWYG, and sound synchronization are performed automatically when called for. When the job is completed the Printer Team notifies the Print Server and suspends itself.

Printer Teams are only active when there is printing to do. If no jobs are being processed, no teams will be active. If more than one job is being processed, multiple Printer Teams may be activated, one for each available printer as in the following diagrams.

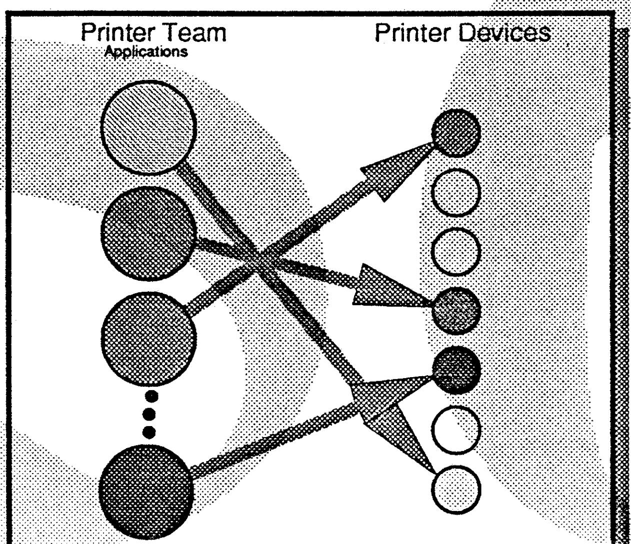


Figure 2. Several Printer Teams can all be operating at once, supporting a number of different printers.

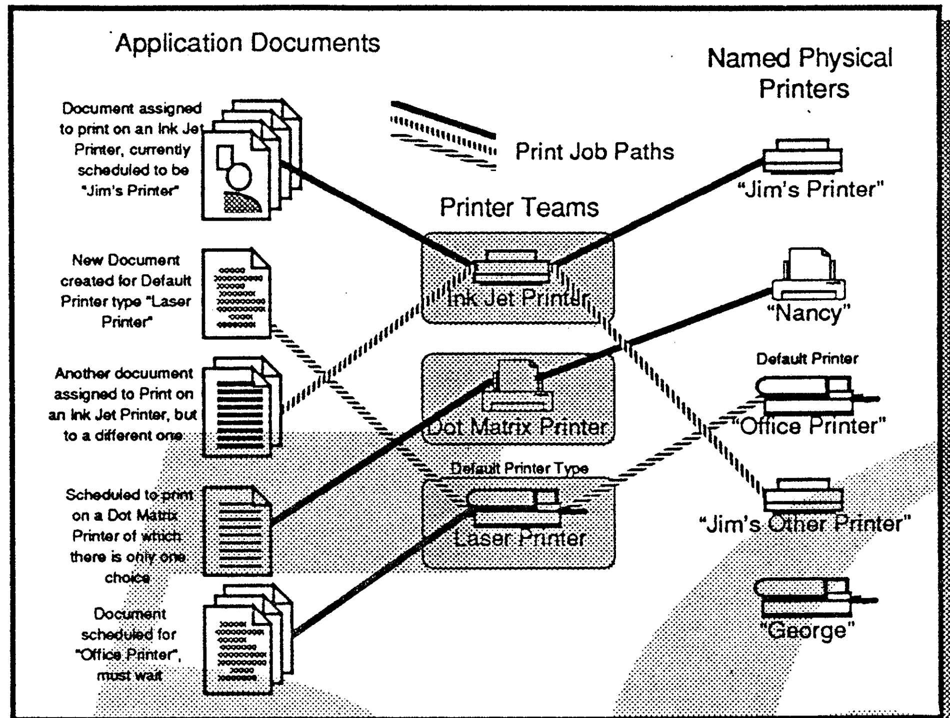


Figure 3. Several Printer Teams operating at once viewed from the print job perspective.

IV. Classes

A Printer Team comprises several component classes. It is at once a member of the TPrinter class, which type it represents, and a client server class. It must also have color matching and a despooling class associated with it.

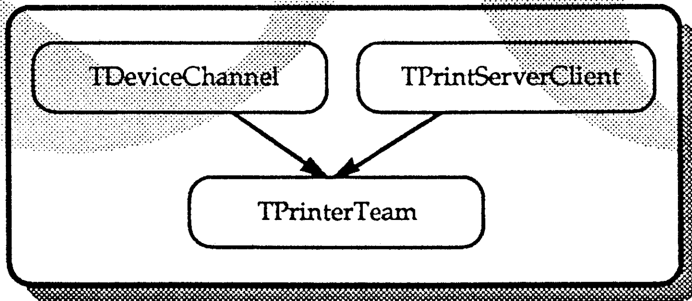


Figure 4. The Printer Team Class.

TDeviceChannel

A device channel is a standard interface for performing I/O between a Printer Team and an actual device.

The device channel is designed to provide a uniform output mechanism for talking to devices. It is designed to allow developers to simulate devices or create virtual devices for preexisting Printer Teams. This makes printing extremely flexible and customizable for almost any conceivable printing application. By this means, Printer Teams can be relatively small and few in number and where the major part of the intelligence is housed. The device specifics can then reside in the TDeviceChannel which can be easily subclassed and modified.

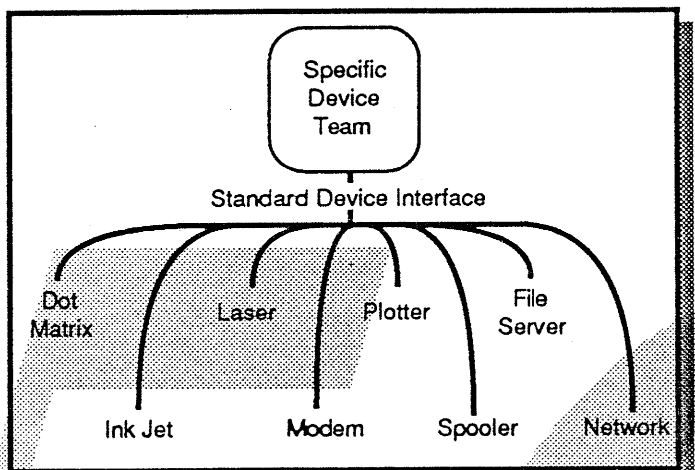


Figure 5. Printer Teams drive printer devices through a standard device interface known as a TDeviceChannel. Theoretically a single Printer Team can be made to support a wide variety of devices interchangeably.

Output from a specific Printer Team can be redirected to any other purpose by connecting into the device channel. The device channel has four separate data streams which can be treated independently and which are used for initialization, data, inquiry, and immediate attention. Normally, this channel implements a simple stream connection between a Printer Team and a specific printer, but it may be subclassed to simulate other purposes, such as secondary transmission to remote printers, testing, simulation, priority access, or whatever. Data that passes through a TDriverChannel interface, however, is not device independent.

The four data streams are all full duplex and completely independent of one another, but have different functions. They are separated into groups for data, inquiry, attention, and initialization. The data channel is for sending and receiving plain vanilla data between the Printer Team and the device. The inquiry channel is used for checking status and access asynchronously. The attention channel also operates asynchronously, but only to abort certain operations, or to alter the state or physical configuration of the printing device. Initialization simply initializes the device.

```
class TDeviceChannel {
public:

        /* The Data Channel */
        virtual size_t              Write(const void* p,long Count);
        virtual size_t              Read(void* p,long Count);

        /* The Inquiry Channel */
        virtual size_t              RequestStatus(const void* p,long Count);
        virtual size_t              RequestAccess(const void* p,long Count);

        /* The Attention Channel */
        virtual void                Abort();
        virtual Boolean             Pause();
        virtual Boolean             Resume();
```

```
        /* The Control Channel */
        virtual Boolean              SendCommand(const void* p,long Amount);

        /* Sound Channel */
 . . .
};
```

A developer may override an existing TDeviceChannel by implementing member functions for all four streams of the channel. The supplied routines are in essence a simulation of a real device but can actually do anything as long as the simulation does not alter Printer Team operation.

TPrinterTeam

The TPrinterTeam makes up the core of the Printer Team. It requests print jobs from the Print Server, prints them and reports on completion.

```
class TPrinterTeam : public MClient {
 . . .
private:
        TPrintSettings              fPrinterAttributes;
        TRGBColor                   fColorMatch;

// Dialog stuff goes in here whenever we can come up with a good dynamic mechanism

public:
                                    TPrinterTeam();
                                    ~TPrinterTeam();

        virtual TJobToPrint*        RequestJob();
        virtual void                JobComplete(TJobToPrint*);
 . . .
};
```

TRecordingGrafDevice

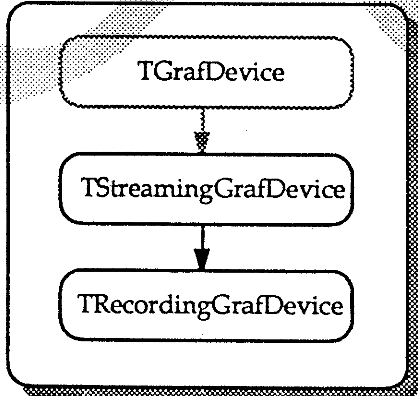


Figure 6. TRecordingGrafDevice is the class used to reproduce aprint job.

This class essentially makes up the print job received from the Print Server. To "despool", the TRecordingGrafDevice is passed to the Printer Team. The Printer Team does this by playing back the TRecordingGrafDevice object itself. A TRecordingGrafDevice has a place to plug in a real printer TGrafDevice to have the recording played into. The Printer Team supplies aTGrafDevice which knows how to play on its own specific type of device.

V. Examples

The Printer Teams primary purpose in life is to sleep. When it is awakened it works only to get back to sleep.

The Printer Team is awakened by some unknown entity, usually the Print Server but that's not important. The Printer Team goes through whatever initialization incantation it needs and eventually ends up requesting a job to process from the Print Server. It prints it using the playback feature and goes back to sleep. It continues to loop until the universe goes away or it is killed by the Print Server.

```
TPrinterTeam        printer;

// initialize and incantate for anything special the Printer Team needs to
// do to the Printer for startup.
• • •

// Request a Print Job from the Server
TJobToPrint* theJob = printer.RequestJob();

// Print the job
theJob.Play(myDeviceGrafPort);
```

At this point the Printer Team has received a TPrintJob that allows it to reconstruct the document, page by page, by replaying it through its own device object. Creation of the printer specific TGrafDevice object is the major job of the Printer Team developer and differs from device to device. This Team can be anything the developer wishes it to be.

To assist developers in writing Printer Teams, there are three tool libraries available that do standard GrafDevice conversions. These are TRasterGrafDevice, TVectorGrafDevice and TPostScriptGrafDevice. Clients can instantiate or subclass any one of these as appropriate to the printer device at hand.

As the data is created for the printer, it is sent to the device over a standard TDeviceChannel.

When the document has been printed, the Printer Team must then notify the Print Server of the status of the document printing process. To register job completion, the client does something like:

```
// Notify the Server
printer.JobComplete();

// Then loop back and do another job
```

If there are no more print jobs for the Printer Team to process, the team just sleeps until one is received.

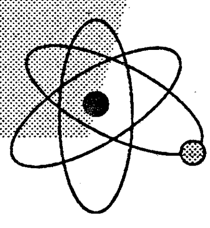
56

Inner Space

56

Scanning

56



Inner Space

Pink Scanning ERS

Bayles Holt
Mahi de Silva
Jay Patel
Ryoji Watanabe

56

Introduction

In 1450, Johann Gutenberg introduced Europe to the printing press and changed the world[1]. In the 1980's, Macintosh helped make desktop publishing a reality[2]. We've become very good at creating hardcopy. We've become *too* good: we are surrounded by much useful graphic and textual information that is available to us *only* in hardcopy form. Even if a piece of information has an accessible digitized form inside the computer, once it is printed out and annotated with a pen it has new information existing only on that hardcopy. When there is a need to interface this information back into the computer, for the typical user today the only available interface devices are the keyboard and the mouse[3].

Of course we can't save the world overnight. It may take up an entire week. The first important and reasonable step is to handle the scanning of *images*. The desktop scanner is our model scanning device. Most of this document (and hence our task) will concentrate on this; we will generalize to broader forms of media input where it is realistic.

Our Goal

Our goal is to increase the role of scanning devices in our users' personal computing experience. We would like to open up totally new applications for personal computers attached to scanning devices as well as accelerate the growing use of scanners for dealing with photographic images. For this, we need to do the following:

- Make it really easy for application developers to access the world of scanned images and graphics.
- Make it really easy for scanning hardware vendors to interface to the Pink system.
- Create a framework that leads developers to implement a user model where scanning is really easy.

Application developers, scanning hardware developers (including Apple), and our users will do the rest.

We won't stop at making scanning easy and inviting. We want to do it right; we need to be concerned with the following:

- WYSIWYG, with one of the important new dimensions being the careful treatment of color.
- Making a design that is extensible or applicable to other kinds of multimedia input.

Purpose of this document

This is the first description of a scanning architecture for Pink. One of the primary goals of this document, therefore, is to warn you about what we are scheming and to invite you to become a critic. But please, no hitting.

---

[1] An ERS that begins like this can't be all that bad... [By the way, my original opening went like this: "The interface between the computer and the hardcopy world is so one-sided that it makes me sick."]

[2] It's turned into a commercial: Run for your lives!

[3] Plagiarism is not enough: We need *computer-aided* plagiarism (C.A.P.).

User's Model of Scanning

Of course, we have the whole user interface worked out already. Unfortunately, it is top secret (yeah... that's the ticket). So we will have to resort to describing things with analogies using elements of the current classic Macintosh interface (declassified circa 1985)...

But to get serious just for one moment: The purpose of this section is to describe the spectrum of functionality that our design provides and to give a general idea of some envisioned user interactions. To give the description some concreteness, we will freely talk in terms of classic Macintosh user interface elements, but these particulars are the least important part of what we want to say.

Some User Scenarios

Meet *Typical User A*. She is a college student working on a 10 page term paper for Art History 101. Being an always well-prepared student, she has already raided the library of all relevant art books containing nice pictures. The first thing she decides to do is scan in all the images she is going to use.

On her personal computer running the Pink system, whenever she wants to scan anything, she always goes to the same place: A standard system software utility we call the Scanning Tool. Imagine this being a small, stand-alone program that is good at doing one thing: Running a scanner. Typical User A might double click on a "Scanning Tool" icon to bring up a single control panel containing various controls for setting scanning parameters, initiating scanning operations, and a graphical "preview" area. She puts the first book on the scanner and hits the "preview" button which does a quick scan and displays a low-resolution image in the preview area. She selects the interesting part of the preview image, chooses a few scanning parameters[4] and initiates a "real" scan. She doesn't worry about matching the geometry of the output image to the destination in her document because she hasn't created that yet. Out pops an icon representing the real image just scanned. Maybe it could be a miniature of the image. She drags it into a scrapbook for safe keeping until she uses it in the paper she will write later. She repeats this process to end up with a dozen images in her scrapbook.

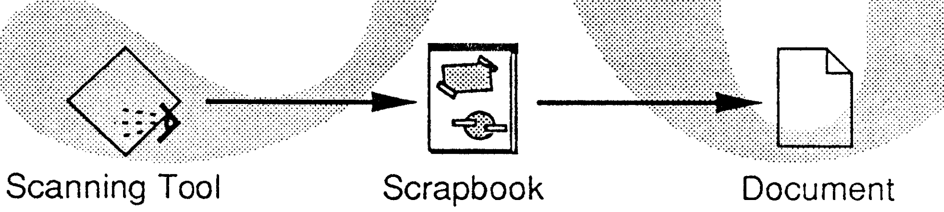


Fig. 1. "Scanning comes first" model

(She figures she only has 5 pages of text to write now.) She is all finished with scanning the library books, but she defers returning them until the next day because Late Night with David Letterman is starting...

[4]How to present the multitude of scanning parameters (like dpi, bit depth, etc.) to a range of users spanning a wide range of expertise is a nontrivial user interface problem.

Meet *Typical User B*. He is in the same art class and has the same paper to write. To his surprise, all the relevant art books (yes, the ones with the nice pictures) in the library are checked out by a person who signed with the name "Typical User A" (no relation to him). Armed with imagination, he creates the text part of the document leaving blank areas for where the images will go. He is so confident that he puts captions on those areas and has text flow around them. The next night, it is his turn with the library books. For him, the scanning process goes like this:

Like Typical User A, he brings up our familiar friend, the Scanning Tool. He also brings up the document that contains the blank areas where the images will be placed. Of course he could just scan the images without considering the geometry of their destinations in the document; the document can do whatever scaling, clipping, or reformating necessary to accommodate them. But he wants to scan it right the first time[5] (for one thing, reformating is out of the question because he wants to preserve his precious page count).

The Scanning Tool has a preview area with a manipulable selection that indicates the "area of interest". What Typical User B really wants to do is make this preview selection the exact same size and shape as the destination area in the document. So he does this in the natural way: He creates a *link*[6] between the preview selection and the destination area in the document; he pushes the area in the document through the link and into the selection in the preview area of the Scanning Tool which then takes on that size and shape. Logically, an empty image is getting passed into the Scanning Tool; the job of the Scanning Tool is to "fill in" this image. He can then position the selection (without changing its size and shape [although of course he can if he wanted]) and do a real scan. The real image can then be pushed back over the same link and into the document.

(There is nothing really special about the link used in this scenario as compared to the classical cut/paste clipboard path. But this example happens to use it well because data needs to be passed in both directions over the same path and the interaction happens over a non-instantaneous period of time.)

After scanning into each of the voids in the document in this way, Typical User B is all done. There is plenty of time to watch Late Night with David Letterman...

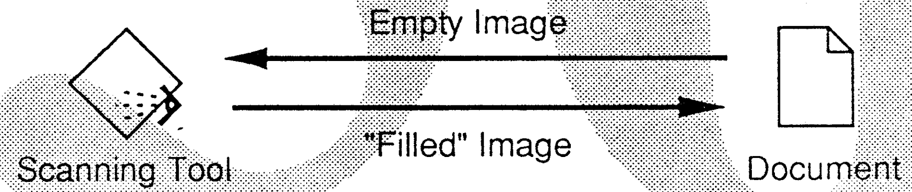


Fig. 2. "Document comes first" model

---

[5]It is always going to be true that one is going to get the *best* results by scanning with the ultimate destination in mind (i.e. avoid unnecessary image processing runs like scaling). There is a human interface problem here as to how to make users aware of this fact of life. One possibility is to present an analogy between taking photographs and scanning: if a person has a lousy photograph, it is already occurs to him naturally that the best remedy may be to take the photograph over.

[6]Part of the CHER document architecture.

Nothing precludes the hybrid case where a user has determined the geometries of the scanned image and the destination independently, i.e. a situation where you have, on the one hand, a scanned image of one size and shape sitting in the Scanning Tool ready to be sent somewhere and, on the other hand, a document with a graphic selection of a different size that is ready to receive the image. When you pass the image from the Scanning Tool to the document's selection, the image *replaces* the selection in the document in the same way it always works when you paste over a selection. What the document does in order to deal with a difference in geometries of the old graphic and the new image is up to the application.

Summary of these user scenarios:

- Users always go to the same facility to scan: the system Scanning Tool (there is no "Scan" command in each application).
- The Scanning Tool can be used independently of any destination document of the images.
- Scanning that is dependent in reasonable ways on the destination of the image is possible and handled in a natural way.

Architecture

Here's a diagram showing the entire design.

Fig. 3. Hieroglyphics

Let's decipher this diagram:

The Scanning Tool is an application ("Tool") supplied by system software that is used to control scanning hardware and produce scanned images. Producing scanned images is all that the Scanning Tool does, so it is very good at it. Although we do not preclude developers introducing specialized Scanning Tools (see Developer's Role below), the suggested model has the user typically turning to the "system" Scanning Tool. See the "Scanning Tool" section below for more details about it.

The user scenarios above described how the Scanning Tool interacts with a document across a link (or cut/paste). This part of the design is built on top of the CHER document architecture— most of the functionality we are getting here is built into CHER and we get it for free. Although we've drawn two paths between the Scanning Tool and the document, data sending via clipboard and data sending via link are very similar. (The scrapbook mentioned in the user scenarios can be considered just another kind of document.[7]) We will happily incorporate into our model future innovations in CHER and the general Pink user interface elements that will go with it.

The parts connected to the Scanning Tool on its left show how it is implemented. The Scanning Tool is a user interface and control center for a scanner. The "real work" of scanning is done by a separate team, the Scanner Team, which reports to the Scanning Tool via interprocess communication. In another world, the Scanner Team might be called a "device driver"; it talks to the scanner hardware directly. It may also include possibly sophisticated image processing— whatever can be called "real work" in a scanning operation is done by this guy. The Scanner Team is a team[8], so it has a lot of resources at its disposal.

The connection between the Scanner Team and the Scanning Tool (the shaded area labeled "standard interface") is invariant. This is so that there can be many varieties of Scanner Teams (to go with the various kinds of scanner hardware) which all "plug in" to the same Scanning Tool. Also, we don't preclude interchangeability on the other end of this connection: There may be specialized Scanning Tools. This is just another way of saying that there may be an application that needs/wants to talk to the Scanner Team directly. If this application is nice enough to make its result of playing with the scanner available via cut/paste/link, then it is truly no different from the system Scanning Tool— it is a specialized Scanning Tool).

How the Scanner Team is implemented beneath the interface to the Scanning Tool is unspecified. Ultimately, we will find commonalties in the many different Scanner Teams for various kinds of hardware so that we will develop a framework for reuse of code through subclassing.

More about the Scanning Tool

The Scanning Tool provides the following:

- Control of scanning and image parameters.
- Control for initiating scanning operations.
- Graphical and direct-manipulation control of output image geometry.
- Capability for dealing with destination image constraints provided to the Scanning Tool *by example* with a blank image passed backward to it from the destination.
- Shows relevant system and scanning status information.

We discuss each of these in more detail next.

---

[7]Note that the document architecture demigods speculate that the Pink clipboard may be capable of holding more than one scrap— it may be more like a scrapbook and make the scrapbook unnecessary.
[8]You have my word on it.

For *scanning and image parameters*, we can list the following:

- Resolution, i.e. "dots per inch".
- Bit depth and halftoning/dithering[9]
- Color model (e.g. grey scale vs color)
- Transparency? — Here's a new one. Since "matte" is part of the Albert graphics model,...
- Output image size, location, shape (i.e. it can be non-rectangular), scaling, clipping.
- Brightness and contrast.
- Filtering, e.g. sharpen, blur.

What will definitely NOT be on this list is a choice of "image file format"[10].

Presenting a multitude of abstract concepts like "resolution", "bit depth", etc. is going to be a nontrivial human interface problem. One way to handle this is to have a HyperCard-like "power user level" setting that each user sets and then have the interface adapt to it.

Scanning operations include the following:

- Doing a "preview" scan.
- Doing a "real" scan.
- Sending scanned images to the clipboard or over links to documents: A single scanned image can be passed to many destinations. In the case where the destination document and the Scanning Tool are connected by a link, the natural interface may be one where the scanned image is automatically sent to the destination.

Graphical and direct-manipulation control of output image geometry will be done in a "preview" area in the user interface. This will include MacDraw-like tools for dragging or drawing out shapes on top of the preview scan image on the screen to delimit the "real" output image.

In the "document comes first" model in our user scenarios, we gave an example of how the Scanning Tool can *deal with destination image constraints*. Although we talked about matching of the *geometry* of a scanned image to its destination in a document. There are other parameters that need to be matched: E.g. bit depth, color model, resolution. For instance, if the receiving document is in high-definition grey scale, then it is likely that the image should be scanned in grey-scale, high bit depth, and high resolution. When the Scanning Tool is passed the "empty" image, it can glean this information (in addition to the geometry) from the blank image. There are many things the Scanning Tool can do with this information. For instance, there may be a "novice" mode where these parameters are hidden. In this case, the tool may choose the parameters based on the blank image. In a "power user" mode where these parameters are visible and manipulable, the blank image may supply the default settings for the parameters.

Status information relevant to scanning include memory (disk) space on the system and an indication whether or not scanning is under way.

---

[9]Digitizing at low bit depths and halftoning/dithering so long before printing (or displaying) time can easily lead to lousy results, so it will be discouraged. However, this functionality will exist for low-memory situations (which should be rarer in the Pink virtual memory environment) and expert users who know what they are getting into.

[10]There are two independent reasons for this: 1) How the image will be "stored" is relegated to the documents receiving the images. 2) The "file format" problem in general will be solved by Pink.

Open Issues

Here are some thing we need to attack.

Economical passing of images. Passing an image over a link or to/from the clipboard semantically involves *copying* the image. In the 98% of cases where the Scanning Tool is used to scan a single image that is immediately passed to a single destination document (and the Scanning Tool immediately throws away its "copy" of the image), we do not want to incur a copy operation in taking the image from the Scanning Tool to the document.

Multiple scanning devices. How will we handle multiple scanning devices attached to a single system? One Scanning Tool for several devices? One to one? Several to one? If we choose to have one (or a few) Scanning Tools for many devices, we will probably implement "personality modules" for Scanner Teams: these are bundles of information that go with each specific Scanner Team (and its associated hardware) that describes anything special about it. One thing is for sure: This is a simpler problem than sharing printing devices because simultaneous use of a scanner is impossible to simulate via a "spooling" mechanism.

Developer's Role

Software Developer

Applications that receive scanned images

What does an application developer have to do to create an application that interfaces in the "standard" way to the world of *scanning*? Nothing! Well, she has to make sure that the application can deal with images at all in the standard way[11], i.e. cutting and pasting. If an application can receive images from the clipboard or a link, then it can receive *scanned* images. If it can put out images, then it can influence the Scanning Tool by providing parameters of the destination image, like it size, shape, and bit depth.

Specialized Scanning Tools

Developers will be able to write applications that talk directly to the Scanner Team. As we discussed in the architecture section, the interface to the Scanner Team is invariant (and the interface to the document is based on CHER and is equally invariant). We hope that such applications are similar in character to the system Scanning Tool in being small utility programs that do just the scanning task (and does it well) and not other tasks that can be separated from it[12], and that it makes its result available to other application via link or cut/paste. We would call such applications "specialized" Scanning Tools.

The various classes that we invent to implement the system Scanning Tool can be made available to developers for use in creating their specialized tools. For instance, the preview window and controls may be something that developers may want to simply "drop in" to their own interfaces.

---

[11]Making a paint program accept scanned images will truly take zero effort; making the Alarm Clock DA take scanned images will take some work, even in the Pink system.
[12]But since we can't prevent Microsoft from entering the game,...

Scanning Hardware Developer

To interface a scanning device to the Pink System, the hardware vendor will create a Scanning Team to go with it. For most mainstream devices, this will probably involve subclassing the classes used to implement a "generic" Scanning Team (provided by system software). Creating a Scanner Team should be easy[13].

Optical Character Recognition

In the case of scanned images, from a document's point of view, a Scanning Tool just appears as just another source of images— it could just as well be another document providing a "cut" image. Similarly, the result of an OCR scan is made available to a document as just another piece of text (or object-oriented graphic).

The CHER link has a type negotiation mechanism where a data source can offer its data in several formats. It would be natural for a Scanning Tool that includes OCR capability to publish a type description that goes like this: "Hey yo document, I have this object and I can provide it as an image, or as a piece of text, or as an object-oriented graphic".

There are several places in the architecture where OCR capability can be plugged in. A software OCR engine might be made part of the Scanning Tool so that it can be used on any image that the Scanning Tool scans. A lower level or hardware based OCR capability might be managed by a special OCR Scanner Team which makes its result available to the Scanning Tool to pass on as usual.

Fax

The scanning architecture finds two applications in a Pink Fax capability. On the one hand, the "fax" object can be something that can accept images via link or clipboard from anyone and transmit it over the phone— it will be just another possible destination for images from a Scanning Tool. On the other hand, there is the task of taking an image received via fax and bringing it to a document. This problem can be solved by making a "fax inbox" object that is similar to our Scanning Tool[14].

Multimedia

Our design can naturally encompass any source of still images. Besides the desktop scanner and the fax, the still video camera and video frame grabber are two more examples. With an expansion of our domain to include entities that deal with time and sound, we can think about an interface to video sources like the camcorder.

There are several properties of our design that we believe should be part of interfaces to other forms of multimedia input:

- The user has one (or a small number) of small "tools" to control the input.
- The interface to the document (CHER link or cut/paste) is one that is widely used by applications.

---

[13]Nay, it should be *fun*.

[14]There are some fundamental differences: Running a "preview scan" is impossible (although it is possible to have a fax inbox tool that allows you to cut out just the interesting part of a received image and throw the rest away), choices concerning resolution, dpi, color model, image size and shape are more concrete.

- Applications do little (no) work to interface to the world of the input media source.

Clearly our design can be extended or applied to any form of multimedia input *where the input information can be packaged (or "recorded") into a static data object.* We might summarize our general design as one that addresses the "media-to-clipboard conversion" problem.

We recognize that there are multimedia sources where it is less appropriate to package the information into a static recording and where the information may better be described as a "continuous signal"[15]. Clearly, in such a case our model is not fit or needs a fundamental extension.

Acknowledgements

David Goldsmith thoughtfully steered us away from an architecture that gave applications more control of scanning but at the expense of extra burden on applications and more complexity. We think he was very right.

The Albert graphics team provides all the "real" graphics machinery to make this all possible. Specifically, Jerry Harris is the Albert image expert who is going to do the "real" work[16].

Arn Schaeffer and Larry Rosenstein is bringing to the world the CHER document model. This is a powerful framework in which we find all the functionality we need for our specific task all laid out already[17].

Jack Palevich continues to be a source of new insights and valuable experience.

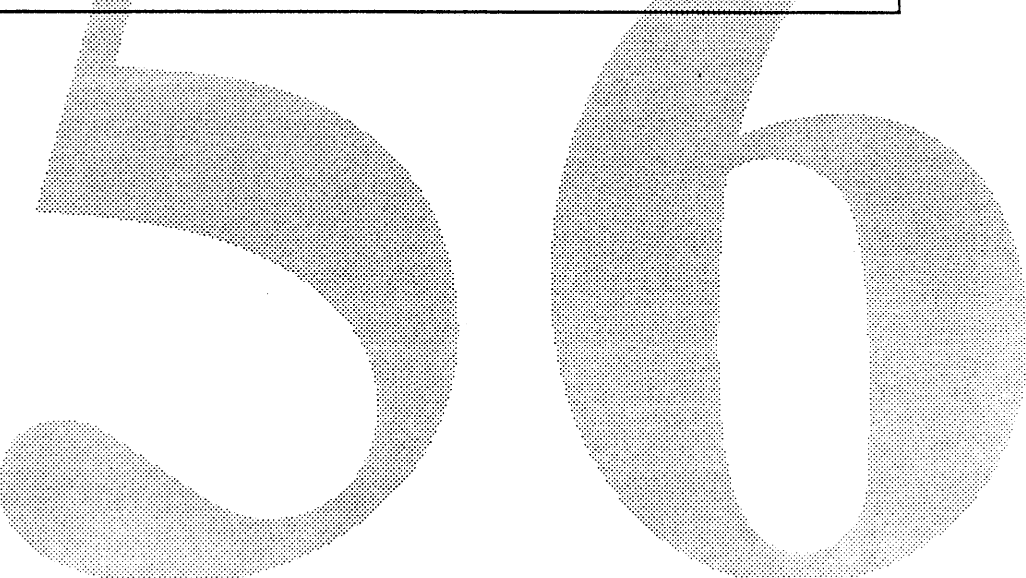
Microsoft Word[18].

[15]Imagine a "TV Channel 4" object that I can paste (or somehow connect) into my document and watch the current broadcast of David Letterman in.
[16]Object-oriented programming tip #11: Step 1: Subclass. Step 2: Take credit.
[17]See previous note.
[18]Just kidding— never mind.

David Letterman's Top Ten Reasons to Scan

10. Scanning is non-pollutionary
9. What else are you going to do with that 300 gigabyte disk?
8. 300 dpi... 150 dpi... 72 dpi... You decide!
7. If we keep 'printing' without balancing it with 'scanning', then the universe gains a net accumulation of positively charged massless Q quarks. We wouldn't want that now, would we?
6. It's (almost) as fun as a Xerox machine, but no one has gotten the idea of charging you to use it yet.
5. It will NEVER run out of toner.
4. In many states (and, in fact, in most nations in the European Common Market and Scandinavia), a picture is *still* worth a thousand words.
3. Halftones for everyone!
2. It's true: a camera *will* steal your soul.
1. **No one will ever know.**

DEI

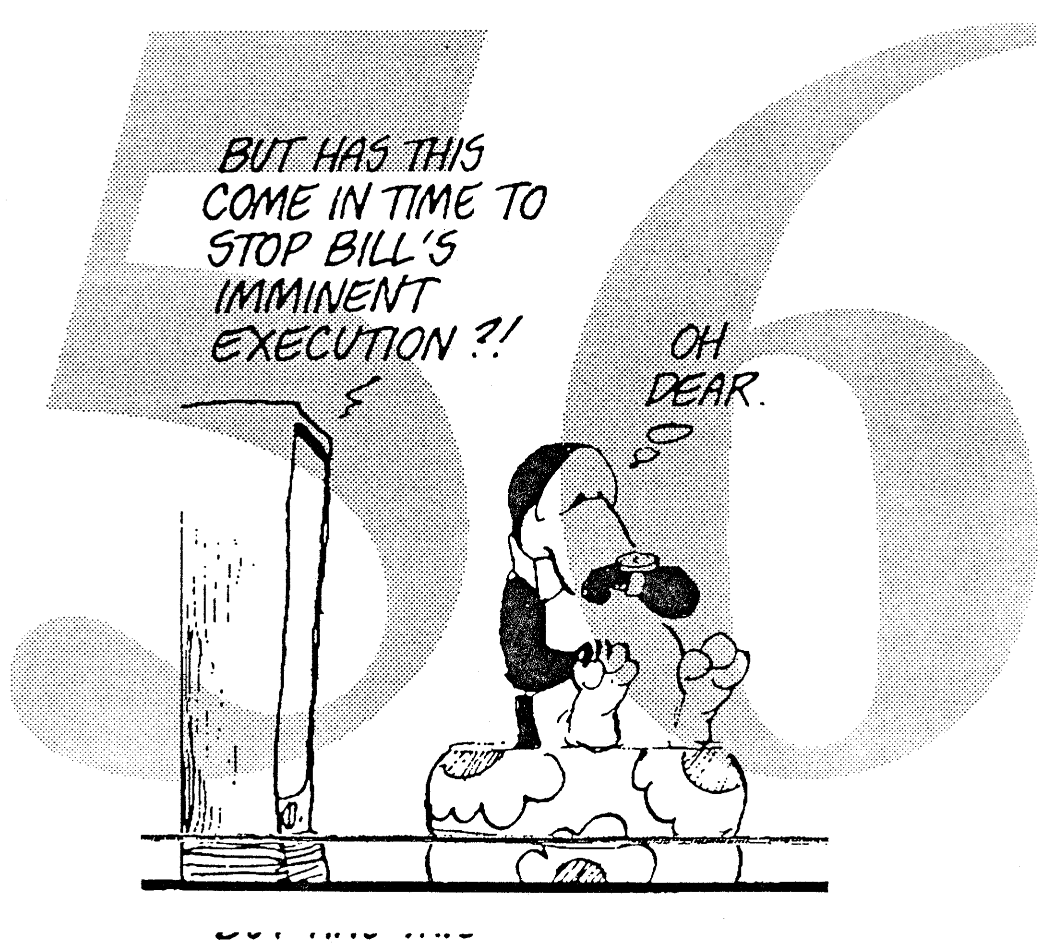


Timing Services

56

Timing Services

Shinzo Watanabe, Patrick Ross, Deb Orton, Dan Chernikoff, Lee Bolton, Matthew Denman, Steve Milne



56

Introduction

This document describes the timing services to be provided by Pink. There are four main components to the timing services, time (TTime), alarms (TAlarm, and TPeriodicAlarm), Date/Time services, and clocks (TClock and TRootClock).

Terminology

Time is a moment or period in some unit of TTime, such as seconds, milliseconds, etc.

Time stamping refers to the action of using a time to stamp, or mark, a piece of data. This is useful when the time that the data was encountered at is important.

A *timing service* is the set of services relating to the setting of alarms for user notification, timing of intervals, time stamping, and basic date/time calculation. This is conceptually different from a *calendar service*, which refers to the services managing calendar and date interpretations. Questions concerning local time zones and standard time vs. daylight saving time, relate to a calendar service.

A *time base* is a representation of time and how it moves. Time bases are implemented using *clocks*. The hardware clock, for example, provides a timebase that runs forward constantly at a regular rate. This can be used for most timing needs. But if there is a need to be synchronized to another representation of time, then the hardware clock will not suffice. The SMPTE (Society of Motion Picture and Television Engineers) clock, for example, is driven by SMPTE time code coming from an external device such as a videotape recorder. The SMPTE clock can stop, change directions, or run at irregular rates - it can even run backwards. All of these cases are likely to happen if someone is manually "shuttling" a video deck. An application that is displaying animation in synchrony with a video deck would use the SMPTE clock as a time base.

Current time is the time (for that time base) at the present.

Time Interval Services

Intervals of time are represented in time objects (TTime). These intervals of time are used by the alarms and interval-timing services to represent various points in time. The time objects allow for the representation of time in many different units, that can be used across different time bases. Time units can also be converted to other time units. It is therefore possible to describe time in units of seconds, milliseconds, microseconds, days, SMPTE frames, samples and more.

Time Interval Model

Time objects provide a convenient way to specify an instance or amount of time and provide a mechanism for performing arithmetic operations on various time objects. Conversions between units (perhaps seconds to video frames) are also provided. Clocks are available for use as a time base, with the hardware clock being the most common (and default) clock. The time object (TTime) is subclassed to provide time in a variety of units (e.g. TMicroseconds, TSeconds, TDays, TSamples, TSMPTE, etc.)

TTime

Methods for General Consumption

Now CREATES a TTime object with the current time. It is a static member function of TTime.
SetToNow fills in the object with the current time.
Delay blocks the task for the specified amount of time.

Methods used in Implementation
GetTime and SetTime set and get the TDoubleLong representation of time.
ConvertTime convert from the hardware representation of time.

TSeconds

(A Sample of the TTime Subclasses.)

Methods for General Consumption
Operator double converts a TSeconds object to a double. Accuracy may be lost.

TStopwatch

This object provides stopwatch functionality. It returns elapsed time since last checked, can be reset and does not have any time drift.

Methods for General Consumption
The **Constructor** creates an object containing the current time.
Reset sets the object to the current time or the specified time.
ElapsedTime returns the amount of time that has elapsed since the stopwatch was last reset.

Alarm Services

Alarm objects provide notification to a task that a certain time has passed. Periodic alarms provide notification repeatedly at certain intervals. Alarms are set on the time base of a clock, and therefore need to specify a clock to use. The default clock is the hardware clock provided by the pink operating system. Each Pink machine has its own hardware clock, which is set at system start-up time.

Alarm Model

A task that wants to make use of the alarm services first queries a clock to get the current time. To set an alarm notification, the application does some arithmetic to determine the time at which it wants the notification to occur, and then creates an alarm object (TAlarm). The task to receive the alarm is specified when the alarm (TAlarm) is created. Once the alarm is set, the task goes on with its regular processing. Eventually, the task to receive the alarm does a WaitForAlarm call (in the case of an MMessageTask) or a Receive call to receive the message associated with the alarm notification. If the notification has not yet occurred, the task will block in the WaitForAlarm/Receive. When the alarm eventually goes off, the clock will send an alarm message (TAlarmMessage) to the specified task. The body of the message will contain the alarm handler object by the task which set the alarm, identification for the clock, and the time of the clock when the alarm went off.

Note that tasks can set alarms whose notification messages are received by other tasks, and a task can cancel any alarm if it knows the original alarm (TAlarm). If the task that sets an alarm is terminated, that alarm is not canceled and will still cause an alarm notification message to be sent to the specified task at the appropriate time. Also, if an alarm is set for a time that is in the past, the notification

message will be sent immediately.

TAlarm

Methods for General Consumption

The **Constructor** creates an alarm object and enables it at the specified time for the specified task.
Equality Operator copies the alarm identifier to the new object. No alarm is enabled (or disabled).
Disable clears the alarm.

Methods for Extending/Changing Behavior

GetAlarmID returns the alarm identifier.
SetAlarmID sets the alarm identifier.

TAlarmHandler

Methods for General Consumption

SetAutomaticallyDestroyed determines if the object deletes itself after the alarm has been processed (and HandleAlarmOccurred has completed).

Methods for Extending/Changing Behavior

HandleAlarmOccurred is called by the receiving task when the alarm is processed.

Methods used in Implementation

IsAutomaticallyDestroyed returns a Boolean telling whether the flag is set.
ProcessAlarm calls *HandleAlarmOccurred* and deallocates the object if *IsAutomaticallyDestroyed* is true.

Interval and Periodic Timing Services

Interval timing services are provided through Stop-watch objects (TStopwatch). Periodic timing services are provided through periodic alarm objects (TPeriodicAlarm).

Interval Timing Model

These objects provide a mechanism for drift-free timing of intervals. This class is based on the time objects described above . A mechanism for starting and stopping the stopwatch, as well as resetting and getting a "lap time", are provided.

Periodic Timing Model

The periodic alarms inherit from the basic alarm object (TAlarm) and export a similar interface. These alarms operate given a start time and a period. Alarms are then generated from the start time and continue to be generated after the given period until the alarm is canceled. Periodic alarms take advantage of light-weight tasks (TAlarmTask) to generate the repeating alarms.

TPeriodicAlarm

This class implements an alarm that automatically fires at theStartTime and then again after

the specified repeat period until the alarm is disabled or destroyed. This class uses a light-weight task to implement the periodic alarm (TAlarmTask).

Methods for General Consumption
The **Constructor** creates and enables the first alarm.
The **Stream Operators** read and write the alarm id to the stream.
Disable clears the alarm..
SetRepeatPeriod changes the repeat period.
GetRepeatPeriod returns the repeat period.
Assignment Operator copies the alarm identifier to the new object.

Methods used in Implementation
GetAlarmID returns an alarm identifier.
SetAlarmID returns an alarm identifier.

TAlarmTask

This class implements a periodic alarm using a light-weight task.

Methods for General Consumption
The **Constructor** creates the task. Start must be called to start execution.
SetRepeatPeriod changes the repeat period of the alarm.
GetRepeatPeriod returns the repeat period of the alarm.

Methods used in Implementation
Main enables an alarm, determines the time for the next alarm and blocks until the next alarm needs to be set.

Clocks

Clocks are the heart of the synchronization and timing services. Conceptually, a clock is just a counter. Clocks are used for creating independent time bases, which can then be used for setting alarms and getting the time. They also provide means for defining a relationship between different time bases.

Clocks can be controlled by sources other than the hardware timer. A clock can be controlled by an external source such as a MIDI clock coming from an Apple MIDI interface, SMPTE code coming from a videotape recorder, or even from an audio object such as a speaker (TSpeaker).

A clock measures the passage of time in fixed units (TTime). Clocks can run at uneven intervals, can speed up and slow down, and can even run backwards (e.g., all of these things will happen when a root clock is synchronized to a videotape unit that is shuttling forwards and backwards).

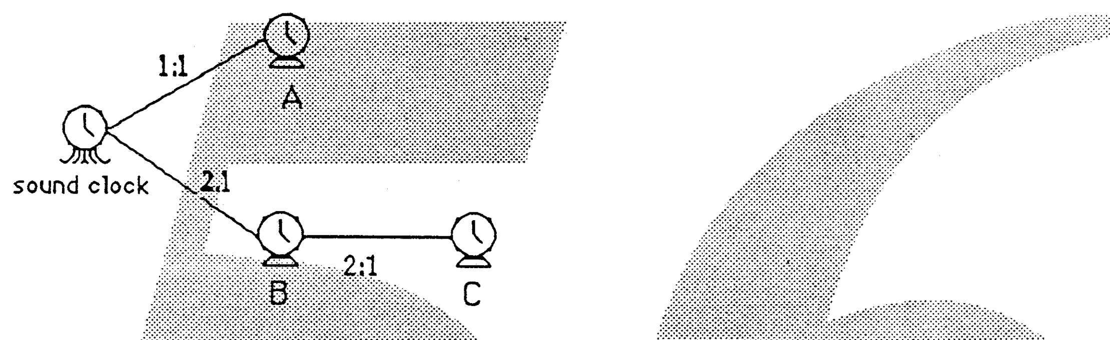
There are two types of clocks: a clock (TClock) and a root clock (TRootClock). A clock is used to represent a local time base which can be used for setting alarms, and getting the time. Root clocks are used to synchronize to a source; such as the hardware timer, SMPTE, or someone's private software counter. One example, is the sound clock which is synchronized to a speaker (TSpeaker). The speaker can set a root clock's time every time samples are played. When a root clock is being set or controlled by a given source, it is said to be synchronized to the source.

TRootClock

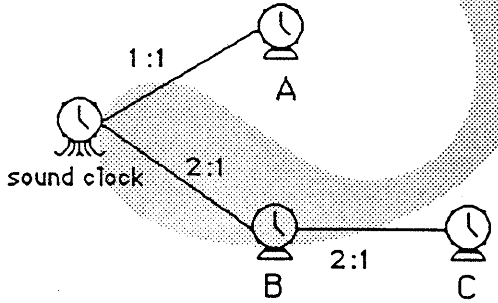
TClock

A clock can be connected to any other clock through a linear function. One of the clocks is the master, the other is the subordinate clock. The master clock can itself be controlled by another clock. In this manner, a chain of clocks can be connected with a defined relationship. A clock can only have its value changed by other clocks. Only a root clock may have its value controlled directly. This is how a new time base can be created and implemented. A clock must be connected (directly or indirectly) to a root clock in order to provide its functionality. Pink will provide some built-in root clocks such as: MIDI clock, sound clock, and the hardware clock.

It is possible then to define a relationship where alarms are set on different clocks all based on the same root clock. For example, three video clips that are played repeatedly, while synchronized to a sound and where the first video clip is to cycle once per sound; the second clip twice per sound; and the third clip four times per sound. The following clocks would be needed, and would need to be connected by the following functions: clock A in a 1:1 relationship with the root clock , clock B in a 2:1 relationship with the root clock, and clock C in a 2:1 relationship with clock B.



Since the clocks are connected in this manner, it would be simple to change the rate of the second and third video clips to be played three and six times per sound, instead. To do this the only change needed would be to change clock B's function to 3:1. Clock C will still run at half the speed of clock B. All of the alarms set on the clocks would still be set to go off at the same local time, but the local time for clocks B and C would be changed at a slower rate.



The abstract relationship between time, alarms, and clocks allows for users to define timing in their own units regardless of their time base. For example, it is possible to write an animation sequence where different characters move at different rates. This may be controlled by setting Alarms in units of milliseconds, and SMPTE time code. This application could run on any pink machine, and could be controlled by any clock, be it a SMPTE clock, hardware clock, or sound clock. This allows for the user to control the synchronization of the animation with other applications.

TClock has methods to get and set its value (the current time), stop the clock, start the clock, and to connect to other clocks. A single TClock may have to be accessible to many teams, probably through the use of CHER. A TRootClock has methods to update its value; this in turn causes the values of all of it's subordinate clocks to be updated. TRootClocks can also be made accessible to many teams through the use of CHER.

MCommonClock

MCommonClock is a mixin class that provides it's derived classes with a common interface for certain clock functionality, such as stopping and starting the clock; changing, getting and setting the direction; and getting the current time. MCommonClock should be used by any Clock related classes that need this functionality, with the same interface. MCommonClock is used by TRootClock and TClock, and should be used in any new clock designs. This class can be accessed simultaneously by multiple tasks.

Methods for General Consumption

GetDirection returns a DirectionState that contains the direction that the clock is pointing in. DirectionState has two possible values, kForwards, and kBackwards. kForwards means that the clock's value is increasing (time is moving forward). kBackwards means that the clock's value is decreasing (time is moving backwards).
SetDirection sets the direction of the clock to the newDirection value, either kForwards or kBackwards. (This also sets the sign of the ratio 'a' for the connecting function, ax + b, of a clock. The sign is + if the dirction of the clock is the same as it's masters, and - if it isn't).
ChangeDirection causes the direction of the clock to change from Forwards to Backwards, or Backwards to Forwards.
Stop stops the clock. Start must be used to restart the clock running again. Stop should only be called when necessary; Clocks will usually run uninterrupted from construction to destruction. Clocks start running when they are constructed, and can only be stopped by a MCommonClock::Stop call, or when destructed.
Start starts the clock running. Usually called only after a MCommonClock::Stop call. Clocks start running when they are constructed, and can only be stopped by a MCommonClock::Stop call, or when destructed.
IsStopped returns TRUE if the clock is stopped, and FALSE if the Clock is not stopped (running).
operator< compares the current time of the clock with a TTime and returns TRUE if the clock is "behind" (less than) theTime , and FALSE if the clock is "ahead" (not less than) theTime . Behind means that the time is in the opposite direction, that the clock is moving in, of the current time. Ahead means that the time is in the direction that the clock is heading. For example, if the clock is at 5 and is going forwards (the next value being 6) then times from 6 to ∞ are considered "ahead". Times from 4 to 0 and 0 to -∞ are considered behind. If the clock was going backwards (the next value being 4) then the time 6 to ∞ are considered "behind", and 4 to -∞ are considered "ahead".
operator> compares the time of the clock with a TTime and returns TRUE if the clock is "ahead" (greater than) theTime , and FALSE if the clock is "behind" (not greater than) theTime . Behind means that the time is in the opposite direction, that the clock is moving in, of the current time. Ahead means that the time is in the direction that the clock is heading. For example, if the clock is at 5 and is going forwards (the next value being 6) then times from 6 to ∞ are considered "ahead". Times from 4 to 0 and 0 to -∞ are considered behind. If the clock was going backwards (the next value being 4) then the time 6 to ∞ are considered "behind", and 4 to -∞ are considered "ahead".
operator== compares the time of the clock with a TTime and returns TRUE if the clock is at the same time as theTime and FALSE if the times are different.

Methods used by TAlarm

Always use TAlarm and TPeriodicAlarm to set alarms. Don't call these member functions directly.

Now returns the current time of the clock.
AddAlarm adds and sets analarm to go off at the specified time.
AddPeriodicAlarm adds and sets a alarm to go off periodically.
Remove removes an alarm.
RemoveAll removes all of the alarms for that clock.

TClock

TClock provides a timebase for setting alarms and timestamping. A TClock must be connected to a TRootClock (directly or inderectly through connections to other clocks that are connected to one). This alows clocks to run in a specifcly defined relationship to each other, and to be able to add alarms, and do timestamping thru a common interface. When a TClock is created it is automatically connected to the system's THardwareClock (which is a specific TRootClock).

Methods for General Consumption

SetFunction sets the function that describes the relationship between the clock and it's master clock. The function must be a linear function, where f(x) = ax + b where 'b' is the offset and 'a' is the ratio. SetFunction effects the time for the TClock, and therefore should only be called when the relationship between the connected clocks changes, and upon initial connection. An Example: If a TClock is being connected to another clock who's current time is 9, and I want my time to start at 0, and run twice as fast the connection function would be 2x - 9. So the offset is -9 and the ratio is 2.
GetFunction gets the offset and ratio of the function f(x) = ax + b of the clock and it's master. 'a' is the ratio, and 'b' is the offset.
SetAtNowPlus sets the time of the clock to the current time (now) plus a TTime.
SetToTime sets the time of the clock to a new TTime.
SetNotificationTask uses the stateNotificationTask task as the task that is called when the state of the clock, or one of it's masters, changes. Changed states of the clock include stopping, changing direction, change in the function between two clocks, and the connection of clocks. This should be called directly if a task will be wanting notification of changes happening to the clock.
SetValueTo changes the offset value of the function describing the relationship between the clock and it's master so that the current clocks value is a new TTime. The offset would be 'b' where the function is f(x) = ax + b. This causes the time to change for the clock. Alarms set prior to the call, go off as if their time was changed by the difference of the current value and the new TTime. For example: if the time was 4 and there was an alarm set to go off at 8. If SetValueTo is called with a new value of 1 (3 less than last offset) the clocks time will be 1 and the alarm will go off when the time reaches 5 (8 - 3 = 5). See SetAtNowPlus and SetToTime for setting the time for the clock without effecting the Alarms. SetOffset should be called whenever the offset, or time of a clock needs to be changed and all previous alarms should change by the same amount.

TRootClock

TRootClock inherits from MCommonClock. TRootClock implements a timebase that other clocks can connect too. To implement the timebase, TRootClock's SetTime method is called to initialize the time for the clock. Then MoveToTimePlus or MoveToTime are called to update the time for the TRootClock. TRootClock is used to be the master of all other clocks connected to it. It is used by TClocks to connect to get different time bases. For example, a hardware clock, MIDI Clock, or Sound Clock, could all be implemented using a TRootClock. TClocks would be used to connect to it, set alarms and do other timing functions. TRootClock always expects to be used with at least one TClock connected to it. TRootClock is controlled by what ever device is providing the interface for the timing base. For instance, a TRootClock could be driven by a animation application that wanted a time base of it's own which it controlled it's own frame rate with. Alarms could then be set for TClocks that where connected to the TRootClock.
Probably only one class will call TRootClock's methods.

Methods for General Consumption

MoveToTimePlus moves the time of the TRootClock from it's current setting to it's current setting offset by the value of a TTime in the root clocks direction. If the plusTime is negative the direction of the clock changes. This call is used to update a TRootClock to a specific time from the current time, all of the functionality of the clock is performed as if the clock was moved thru time to the new value.

All alarms between the current setting and the new setting will go off. For example if the clock is at 0 moving backwards, and a plusTime of -3 is provided, the direction is changed and the time is moved to 3 (the clock will be moving forwards).
MoveToTime moves the time for the root clock to a new TTime. This call is used to update a root clock to a specific time, all of the functionality of the clock is performed as the clock is moved thru time to the new value. If the newTime is in the opposite direction of the TRootClock's direction, the dirction of the TRootClock is Changed. All alarms that were added for the time between the current setting and the new setting will go off.
SetTime sets the time of the root clock to a new TTime, and removes all alarms set for that root clock by any of it's subordinates. This call should only be made when it is necessary that the RootClock is set to a specific time, or needs to be reset or initialized. This call is not to be made to update the time for the clock.
GetResolution returns the resolution of the clock. This call should be used to help determine how accurate a clock is.
SetResolution is used to set the resolution of a clock. The resolution is the largest amount of time that the clock may move through at one time. This call should be made to set the resolution which can provide useful information in determining the accuracy of a clock.

Date/Time Services

Date/Time objects provide the number of days (in some resolution) since the Julian date (Days since 4713 B.C. noon) in Universal time as specified by the hardware date clock. There is no interpretation of the calendar date, as "time" is presented only as a count of the number of days since the given origin.

Date/Time Model

The date/time classes export the Julian Date in seconds. This is a count of the actual number of day/night periods that have passed since a given starting point. The date/time classes are implemented using an object representing a date (TDateTime) and an object representing the difference between two dates (TDateSeconds). Arithmetic operations may be performed on date/time objects, as well as conversions between date/time intervals and time objects (TTime). Calendar interpretations of the date/time objects will be provided by other classes.

TDateTime

Constants
kJulianOriginDate number of seconds since January 1st, 4713 BC at noon.
kMacOriginDate number of seconds since January 1st, 1941 AD at midnight.
kModifiedJulianOriginDate the modified Julian date.
kUnixOriginDate number of seconds since January 1st, 1970 AD at midnight
kDosOriginDate number of seconds since January 1st, 1980 AD at midnight

Methods for General Consumption
SetToToday fills the object with the current date.
Today is a static member function which CREATES an object containing the current date.
operator double convert to a double containing the number of seconds.
operator TDoubleLong convert to a TDoubleLong containing the number of seconds.

Methods used in Implementation

HardwareToJulian convert from hardware representation to the Julian date.
SetHardware initialize the hardware to the specified date.

TDateSeconds

This object exports all the TDoubleLong operators and provides conversions between itself and the built in type "double". There are also conversion operators for TTime objects.

Notes

A little known, but highly useful method is provided called ***TwiddleMMU()****. This method allows unrestrained twiddling of the MMU (Memory Management Unit) and is included as part of the alarm interface because the OS weenies found it alarming. Twiddling the MMU is especially useful when frustration levels are high and productivity low or when you have nothing better to do.*

56

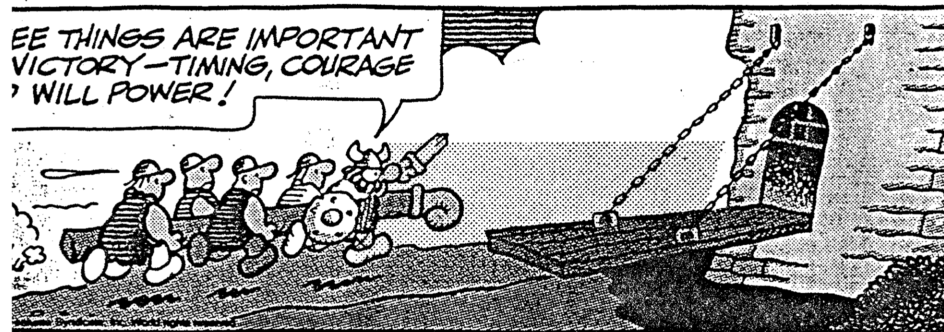
Time Ports and Sequences

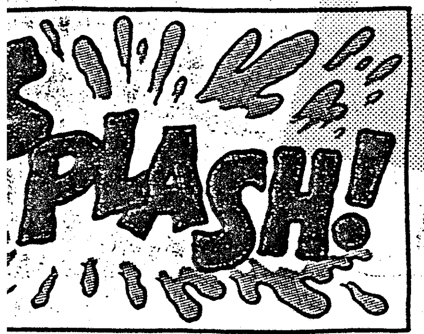
56

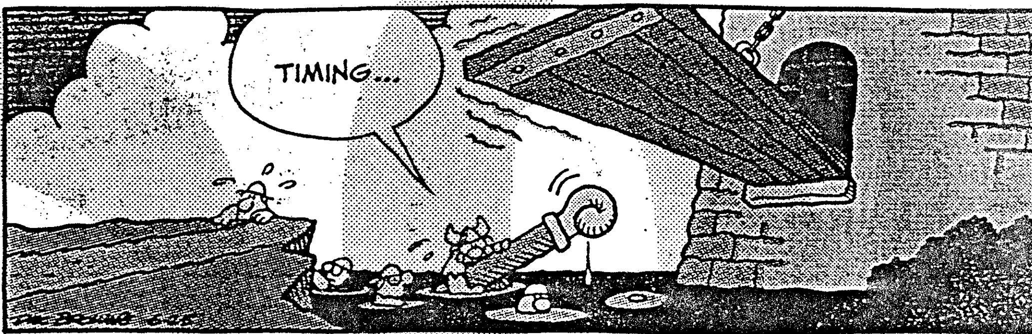
Time Ports and Sequences

Matthew Denman, Steve Milne

r the Horrible □ Dik Browne







56

Architecture

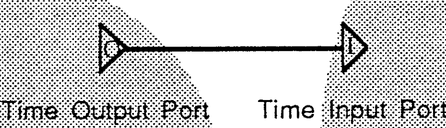
Time Ports and Sequences are used to help control the flow of time-related data. Examples of time-related data are 1) MIDI (Musical Instrument Digital Interface) data, which is used to control music synthesizers; 2) a series of frames in an animation; or 3) any data that is ordered in time.

Time Data

Time data (TTimeData) is a class that consists of two important objects - a time stamp and a chunk of data. The timestamp is a TTime object. The data is a TMemory object. TTimeData has methods to create, modify and compare time data. Time data can be used to hold MIDI commands, other non-MIDI representations of musical data, points of interest in an animation, or any other time-related data. The data part of a time data object must be understandable on its own. It must be intelligible even if interleaved with other data of the same type. For example, you cannot use the data to represent a MIDI command with free running status.

Time Data Ports

Time data ports (TTimeDataPort) are used to send and receive time data. Events are written to output time ports (TOutputTimePort) and can be read from a connected input time port (TInputTimePort). TInputTimePort and TOutputTimePort are descendants of TTimePort. Time data ports can use a clock (TClock) as the timing source for the port. Output ports can be connected to input ports, probably through CHER.



Input Time Data Ports

All time data objects that are read by an input port have a time stamp of the current time of the clock it is using, when the data was sent. If the input port is not using a clock, then a time of zero is used. Time data may be read from an input port by a blocking read, or a non blocking read.

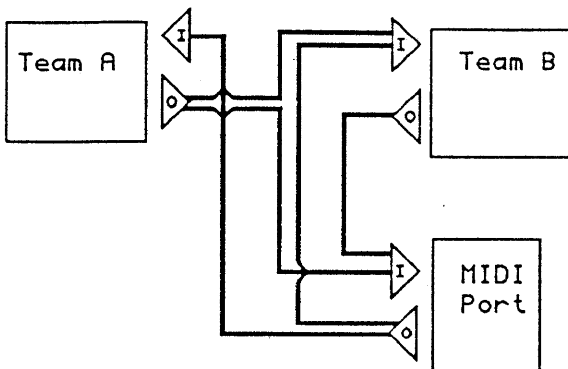
Output Time Data Ports

An Output port in one team can write to an Input port in another team. TOutputTimePort has functions for using a clock, and writing time data. Time data may be written to an output port by a blocking write, or a non blocking write. The blocking write will block the task until the time data has been received by all of the input ports it is to be sent to.

Time ports can be connected to each other over a variety of different teams. There can be multiple connections from one output port going to many different input ports. There may also be many output ports that are all connected to one input port. Data flows from one port to another when an output port is written to. That data is then transmitted to any input port it is connected to. The teams that own those input ports can then read the data.

An Example - MIDI

MIDI data needs to have its time preserved by a time stamp, so that the rhythm information is retained. The MIDI data also needs to be sent at the appropriate time. A user may want to have MIDI shared between multiple teams.



In the above diagram, the MIDI port uses the serial port hardware to communicate with an external music synthesizer. Both teams A and B receive any data coming in from the MIDI port. This data is time stamped for them by the clock used with their input ports. Team B is also receiving data from Team A's output port. This data will also be time stamped by Team B's clock. Team B is sent data from it's output port to the MIDI Port's input port. This data is sent when Team B's clock is equal to the time stamp for the data to be sent. Team A is also sending data when it's output port clock says it is time to be sent. Team A's data is sent to both Team B and the MIDI Port's Input Ports.

Time Port Editor

In MIDI applications, it is desirable to visually connect time ports together using a graphical editor. Such an editor is supplied with the MIDI Management Tools currently provided by Apple for the Macintosh. This editor, called PatchBay™, graphically depicts applications and their MIDI time ports. End users can patch the time ports together in any manner they please. Such an editor is an intuitive and powerful tool for musicians, who are used to patching physical MIDI cables together. There will be some similar way of doing this for all time ports in Pink, probably using CHER.

Sequences

Time events can be collected into an ordered list called a sequence. Each time event in a sequence can mapped to any output port for that team. When a sequence is played, the events are sent out their appropriate output time data ports at their scheduled time. A Sequence can also use multiple input time data ports. The data from these ports can be recorded directly into a sequence. Sequences will not be implemented for the first release of Pink. Possible uses of Sequences would be for implementing a MIDI sequencer, or an animation sequencer that runs MIDI, animation, and sound effects.

MIDI

Pink provides a MIDI Driver that works with the Apple MIDI Interface and compatible Interfaces. This driver has two data ports, one for input one for output. Time data objects that contain a MIDI command for the data can be written to the input port, and will be sent to the MIDI Interface. MIDI information coming in from the MIDI interface will be packaged into time data objects as whole MIDI commands, and will be time stamped using the MIDI clock provided with Pink. Multiple output ports may be connected to the MIDI input port, and the MIDI output port may be connected to multiple input ports.

Classes

TTimeData

TTimeData is the main object used with TimeDataPorts, it is the object that is passed between data ports. It contains two parts: a time stamp (TTime), and a chunk of data (TMemory). It has methods for getting and setting the time and data. It also has methods for comparing it to other TTimeData objects. It should be used whenever use of data ports is desirable.

```
class TTimeData: public TTime, public TMemory{
public:
      MPersistentMacro(TTimeData);
                          TTimeData();
                          TTimeData(const TTimeData& aTimeData);
      virtual             ~TTimeData();
      virtual void        SetTime(const TTime& theTime);
      virtual TTime       GetTime();
      virtual void        SetData(const TMemory& theData);
      virtual TMemory     GetData();
      virtual Boolean     operator> (TTimeData& aTimeData);
      virtual Boolean     operator< (TTimeData& aTimeData);
      virtual Boolean     operator== (TTimeData& aTimeData);

};
```

```
void TTimeData::SetTime(const TTime& theTime);
```
Sets the TTime of the TTimeData object to theTime.

```
TTime TTimeData::GetTime();
```
returns the TTime of the TTimeData.

```
void TTimeData::SetData(const TMemory& theData);
```
sets the TMemory of the TTimeData object to theData.

```
TMemory TTimeData::GetData();
```
returns the TMemory of the TTimeData.

```
Boolean TTimeData::operator> (TTimeData& aTimeData);
```
Compares the time of the object with the time of aTimeData object. It returns TRUE if it is greater than the 'aTimeData'.

```
Boolean TTimeData::operator< (TTimeData& aTimeData);
```
Compares the time of the object with the time of aTimeData object. It returns TRUE if it is less than the 'aTimeData'.

```
Boolean TTimeData::operator== (TTimeData& aTimeData);
```
Compares the time of the object with the time of aTimeData object. It returns TRUE if the two times are the same.

MTimeDataPort

MTimeDataPort is a mixin class used to provide common functionality for the different kinds of time data ports. It has methods for flushing events and getting and setting clocks.

```
class MTimeDataPort{
```

```
public:
      MPersistentMacro(MTimeDataPort);
                        MTimeDataPort();
      virtual           ~MTimeDataPort();
      virtual void      SetClock(const TClock& portsClock);
      virtual const TClock&   GetClock();
      virtual void      Flush();

};
```

```
void MTimeDataPort::SetClock(const TClock& portsClock);
```
Causes the portsClock to be used by the data port for all of it's timing needs.

```
const TClock& MTimeDataPort::GetClock();
```
Returns the clock that is being used by the data port.

```
void MTimeDataPort::Flush();
```
Discards all TTimeData objects that are currently being buffered. TTimeData objects are buffered when they have been written to a port, but not sent yet; or when they have arrived at input ports but not been read yet.

TInputTimeDataPort

TInputTimeDataPort descends from MTimeDataPort and is used for inputting TTimeData objects from a TOutputTimeDataPort to a team. It provides two methods for reading TTimeData objects; a blocking read (ReadTimeData) and a non blocking (PollTimeData).

```
class TInputTimeDataPort: public MTimeDataPort{
public:
      MPersistentMacro(TInputTimeDataPort);
                                    TInputTimeDataPort();
      virtual                       ~TInputTimeDataPort();
      virtual const TTimeData&      ReadTimeData(); // Blocking
      virtual Boolean               PollTimeData(const TTimeData& readTimeData);
      virtual void                  SetTaskForReceive(MBaseTask& receiveTask);
};
```

```
const TTimeData&  ReadTimeData();
```
ReadTimeData is a blocking read of the port. It will return a read and return a reference to the next available TTimeData. If one is not immediately available, it blocks the task until one is available. If the port is using a clock the TTimeData object is time stamped with the clocks value when it is read. If a clock isn't being used a time stamp of zero is used.

```
Boolean PollTimeData(const TTimeData& readTimeData);
```
PollTimeData is a non blocking read of the port. It will put a reference to the next available TTimeData in readTimeData if one is available. If one is not immediately available, PollTimeData returns FALSE; it returns TRUE if one was available, and is now referred to in readTimeData. If the port is using a clock the TTimeData object is time stamped with the clocks value when it is read. If a clock isn't being used a time stamp of zero is used.

TInputTimeDataPort

TOutputTimeDataPort descends from MTimeDataPort and is used for outputting TTimeData objects from a team to the TInputTimeDataPort of another team. It provides two methods for writing TTimeData objects;

56

a non blocking write (SendTimeData) and a blocking (WriteTimeData). Most users will want to use SendTimeData.

```
class TOutputTimeDataPort: public MTimeDataPort{
public:
      MPersistentMacro(TOutputTimeDataPort);
                                    TOutputTimeDataPort();
      virtual                       ~TOutputTimeDataPort();
      virtual void                  WriteTimeData(TTimeData& output); // Blocking.
      virtual void                  SendTimeData(TTimeData& output);
};
```

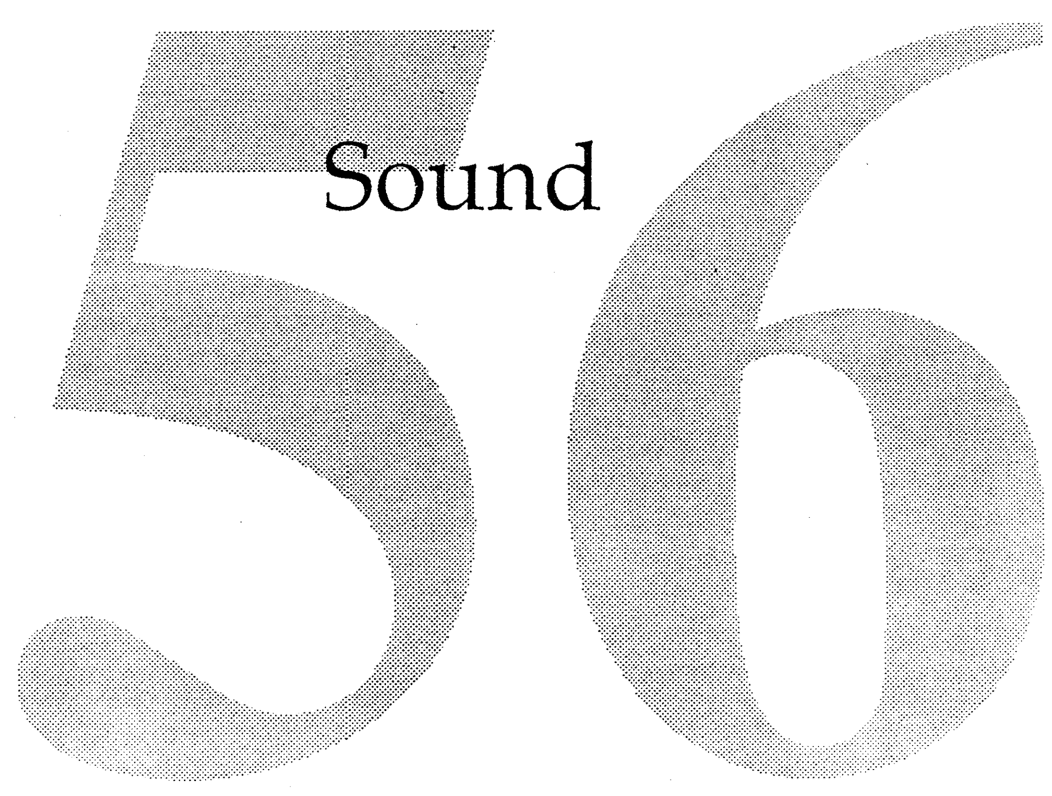
`void WriteTimeData(TTimeData& output);`

WriteTimeData is a blocking write of the TTimeData output object. Output is either sent immediately if there is no clock being used, or is buffered until the time of the TTimeData object is equal to or less than that of the clock being used. When the data is sent, it is sent to all of the TInputTimeDataPorts that the TOutputTimeDataPort is connected to. The current task is blocked until the TTimeData object has been read by all of the TInputTimeDataPorts that it was sent to. This method should be used only when writing TTimeData objects, where the speed and timing of sending the data is unimportant, but where there may be input ports that may need to receive data at a rate which insures that they can process it as it comes in. The task may be blocked for a long time if any of the input ports are very slow readers, or aren't reading the data at all.

`void SendTimeData(TTimeData& output);`

SendTimeData is a non blocking write (or send) of the TTimeData output object. Output is either sent immediately if there is no clock being used, or is buffered until the time of the TTimeData object is equal to or less than that of the clock being used. When the data is sent, it is sent to all of the TInputTimeDataPorts that the TOutputTimeDataPort is connected to. The current task is not blocked and returns immediately after TTimeData has been sent, or buffered. This method should be used for sending TTimeData objects, unless there is some need to insure that input ports are keeping up.

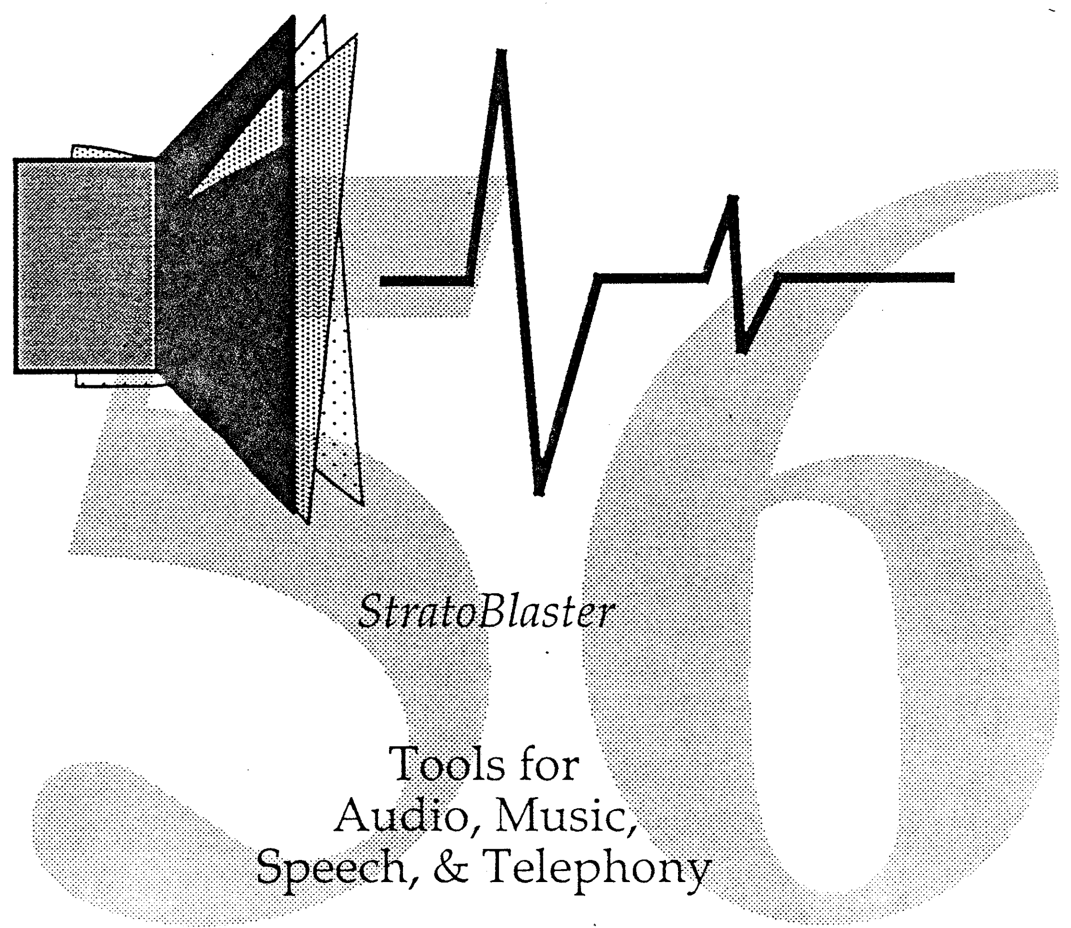
56



Sound

56

Pink Sound Tools



StratoBlaster

Tools for
Audio, Music,
Speech, & Telephony

Steve Milne
Matt Denman
Mike Dilts
Bill Aspromonte

With help from:
Jim Nichols,
Mark Lentczner, Eric Anderson,
the Pink Team. and many others.

56

Introduction

The Pink Sound Tools allow developers to build applications using audio, music, speech, and telephony. The tools consist of a set of C++ class definitions and various hardware specific implementations for those classes.

This section describes the architecture of the sound tools. The meaning of the word "architecture" refers to the design philosophy and a high-level description of the various objects, how they relate to each other, and what they accomplish. The programmer's view of the tools is the main focus of this section.

This section does not detail the definition of each member function of every object. The complete C++ definitions of the objects are found in the Pink On Line Class descriptions. The architectural ideas should remain relatively stable even if the details of the C++ object definitions change.

There are an ambitious number of classes described in this section, more than will be ready to ship with release 1.0 of Pink. This architecture is meant to span a one to four year implementation period. This dovetails nicely with our hardware plans, because many of the objects can only run in real-time on future machines that have more CPU power than today's platforms. Digital signal processors (DSPs) or reduced instruction set (RISC) processors can provide the required power.

Design Goals

1. End-user focus. We need to give developers the tools they need to write sound applications that satisfy customer's needs or wants. Our expectations are that customers will want to use the following sound applications:

- voice annotation
- voice mail
- soundtracks for multimedia presentations and animations
- telephone control
- remote telephone access to electronic mail and other information
- voice control of computers (speech recognition)
- sound feedback in the human interface
- music synthesis and composition

2. Standard Application Programmer Interface (API). Developers need a standard API for audio, speech, and telephony so their applications will work across a wide variety of different sound hardware platforms. We need to provide for *hardware independence.* A phone answering application, for example, shouldn't have to concern itself with what type of phone system the user's computer is connected to. The tools should allow the application to work with any phone system as long as the customer purchases the appropriate hardware and object libraries.

3. Simplicity. Sound is a new technology for many developers. The tools have to be easy for developers to use, or they won't use sound at all. To borrow an overused but apt maxim, "Simple things should be simple, difficult things possible."

4. Generality. Because sound in personal computing is so new, it is impossible for us as toolsmiths to predict exactly what the big applications for sound will be. Therefore design must emphasize generality and flexibility, so developers can create applications we didn't think of.

5. Extensibility. Developers will want to add new functions to our tools. Luckily, extensibility can be ac-

complished easily with C++ by subclassing objects.

6. Scalability. A given sound application should run across the widest possible variety of machines. This is difficult with sound, due to its real-time nature. A speech recognition algorithm that consumes 90% of a RISC processor's CPU time will not run on a Mac SE/30. Other algorithms can be degraded, however. For example, when mixing multiple sounds together, some sounds could be dropped. Where possible, we must provide *graceful degradation.*

7. Synchronization. Often sound playback is most useful when combined with animation or video. We must provide a means for synchronizing sound to other system functions and visa versa. The sound tools use the Pink Timer Tools (chapter 2.5) as a basis for synchronization.

8. Standard interface for sound. We need to provide a standard user interface for recording, editing, and playing sounds. Such an interface would define the way that users edit sound, much like TextEdit did for text on the Macintosh. An editor can also be used by developers to create and edit sounds.

9. Standard interchange formats. Users will want to trade and swap sounds. Our tools must provide standard formats for storing audio data.

10. Reliability. Our tools must work. Because sound is a real-time process, it will exercise the system in different ways than most standard applications. Multitasking must be taken into account, as multiple processes will want to make sounds simultaneously. This puts a burden on the sound tools and Pink System designers to insure that sound performs properly, so that developers can rely on it.

Overview

The Pink Sound Tools documentation is divided into four sections: Audio Objects; Sound, Speech, and Telephony; Editors; and Sound Effects.

Audio Objects (2.6.1) describes the architecture of MAudio, from which audio, speech, telephony, and music classes descend.

Sound, Speech,and Telephony (2.6.2) describes classes that the programmer can use for creating audio, speech, telephony, and music applications.

Editors (2.6.3) describes the classes used for sound editors. Editors provide a standard user interface for creating, editing, and playing, sound.

Sound Effects (2.6.4) discusses the sound effects library that will have to be shipped with Pink.

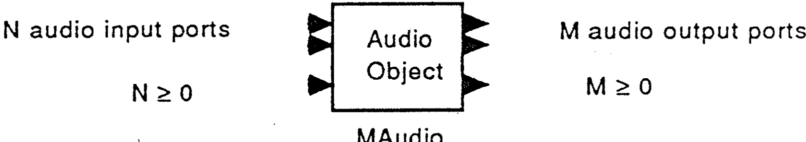
56

Audio Objects

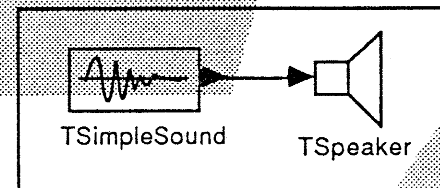
56

Architecture

Audio objects are the heart of the Pink sound tools. Audio objects generate, process, or consume audio data. All audio objects are descendants of the C++ class, MAudio. An audio object can have N audio input ports and M audio output ports.



Audio objects can be connected together by connecting their ports. This is analogous to using patchcords to connect audio components together in the real world. In the illustration below, an audio object, TSimpleSound, is connected to a speaker, TSpeaker.

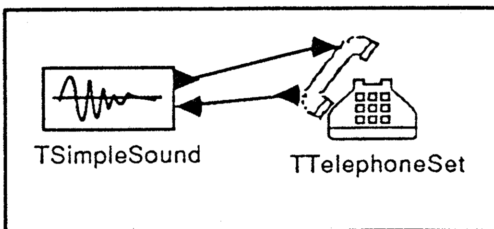


Audio objects are controlled using C++ member functions. TSimpleSound, for example, has member functions for playing and recording audio from disk files. When TSimpleSound's member function Play() is called, audio data will be fed from TSimpleSound to TSpeaker, which will cause the audio to be heard on the computer's speaker. TSpeaker does not have a Play() function, because it just passively plays whatever audio data is pumped into it. TSpeaker does have a SetVolume() control, however, which sets the volume for whatever data plays through it.

Generally, audio objects are implemented completely in software. However, it is possible for an audio object to represent a physical piece of hardware, such as TSpeaker represents the playback hardware of a computer. In this way, external audio devices, such as third party plug in cards or external boxes, can be represented as audio objects.

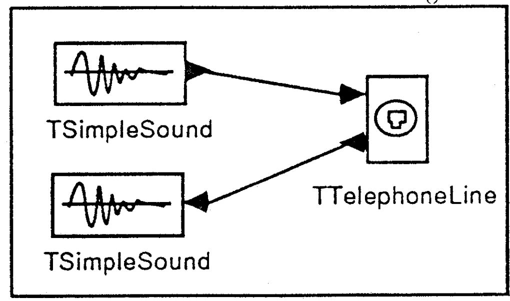
Audio objects can easily be combined to create a variety of interesting sound applications. Below are some examples.

Voice Annotation



TSimpleSound both records and plays voice annotations. The telephone handset is used for input and output.

Voice Mail/Phone Answering



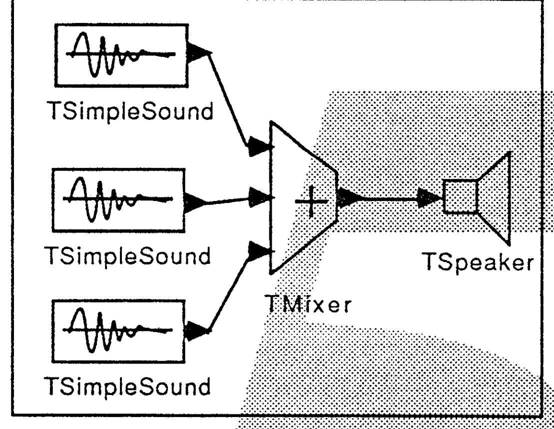
One sound plays the greeting. The other records a message.

Music Synthesis

TSampler TReverb TSpeaker

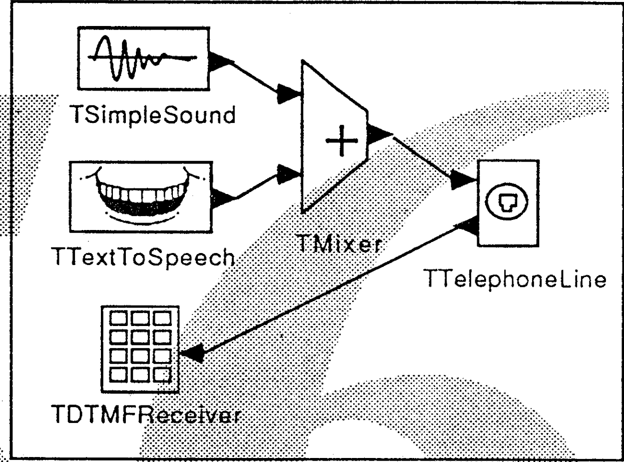
Reverb is added to a sampling synthesizer before it is played through the speaker.

Multi media soundtrack



Three sounds, one for music, one for sound effects, and one for voice-over, are mixed together and feed through the speaker.

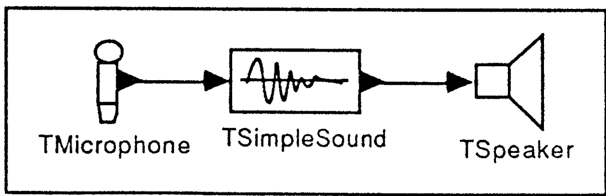
Remote Voice Access



Voice messages are recorded and played with TSimpleSound. Text messages are read with a text-to-speech synthesizer. Touch tones, for user control, are recognized with the DTMF (Dual Tone Multi-Frequency) receiver.

Simplicity

Audio objects allow great flexibility for handling a wide variety of applications. However, many programmers will want to do simple things - such as playing a sound - without having to instantiate a lot of objects and connect them together. The Pink Sound Tools provides "convenience" audio objects for those functions that are needed often. One such object, TSampledSound, is a single object useful for recording and playing sound. TSampledSound conceptually contains a TMicrophone, a TSimpleSound, and a TSpeaker.



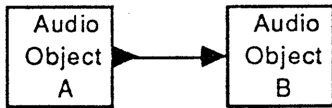
TSampledSound

Playing a sound then becomes as simple as:

```
TSampledSound   beep("/Sounds/Beep");      // Instantiate a beep sound.
beep.Play ();                              // Tell it to play.
```

Connecting Audio Objects

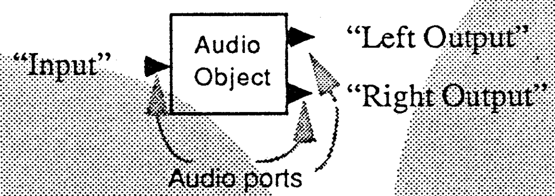
Audio objects are connected together by connecting one of the output ports of one audio object to an input port of another.



The connection is unidirectional. In the above illustration, audio flows from audio object A to audio object B. Audio object A is said to be the *producer*, while audio object B is the *consumer*.

Audio Ports

The C++ class TAudioPort is used to represent audio ports. Both Input and output ports are named using either strings or tokens (TToken). An audio object has a list of audio input ports and a list of audio output ports.



The TAudioPort:: ConnectTo () member function is used to connect the output port of one object to the input port of the other.

```
TSimpleSound        a;
TSpeaker            b;

a.GetOutputPort("Output").ConnectTo(b.GetInputPort("Input"));
```

The MAudio::GetOutputPort() searches an audio object's output port list looking for the requested port. It returns a TAudioPort if a match is found, otherwise it generates an exception. MAudio::GetInputPort() does the same for an audio object's input port list. Both of these functions are overloaded to accept TToken objects also.

TAudioPort::Disconnect() is used to disconnect ports.

Audio Types

Each audio port has an *audio type* associated with it. The C++ class TAudioType is used to represent audio types. Audio type specifies what the format of the audio data is. An audio type consist of three fields, *data format, sample width,* and *sample rate.*

Data format specifies the type of samples - offset binary, linear, or floating point. Sample width determines how many bits are in each sample. Sample rate measures the number of samples per second. Typical audio types are listed below:

Audio Type			Description
Data format	Sample Width (bits)	Sample Rate (samples/sec)	
Offset Binary	8	22,254.54	Native to Macintosh
Linear	16	44,100	Used by CD players, integer DSPs.
Float	32	48000	Native to RISC, floating point DSPs

When an audio output port is connected to an audio input port, the audio types must match. Otherwise, an exception will be generated. This requires that converters be available to convert between the various formats and sample rates. Converters are themselves audio objects. The Pink Sound Tools will provide a general purpose converter, TConverter, which will convert from one audio data type to another. A TConverter has one input port and one output port. When a converter is created, it is passed parameters to indicate what type will be converted to what. That way, the type of the converter's ports is known when the ConnectTo() call is made.

Conversions between types, especially between different sample rates, can be computationally expensive. The number of different audio types used on a single computer will need to be minimized so that type conversions do not consume too much real time.

New data formats should be definable in the future by Apple and third party developers.

The TAudioPort::ConnectTo() may also fail because there is no hardware support for the connection in question. For example, if a connect call is made to a TTelephoneLine, and there is no telephone hardware on the computer, then the call will signal an exception.

Audio Players

An *audio player* is an audio object that represents a sound. Audio players have a Play() member function, so that the sound can be heard. Some examples of audio players are sound files, music synthesizers, speech synthesizers, and tone generators. Because this type of object is so common, there is a mixin class defined for it - MAudioPlayer. MAudioPlayer provides an abstract interface for playing, stopping, and navigating around in streams of audio.

Defining an abstract base class for audio players has the benefit of making it possible to play any object polymorphically, as long as it descends from MAudioPlayer. For example, it is possible play each sound in a list of pointers to MAudioPlayers without caring if the sound is a sampled sound, synthesized musical note, sound effect, or segment of synthesized speech.

The most important member functions of MAudioPlayer are Play(), which starts playback, and Stop(), which stops playback. GetPosition () returns a TTime containing the current play pointer into the sound. This is analogous to the value of the tape counter on a tape recorder. GetPosition can be called after Stop() to determine where playback stopped. Another function, SetPlayRange(), takes two TTime objects as parameters. It restricts playback to the range in the sound between the two TTime objects. These four functions - Play(), Stop(), GetPosition(), and SetPlayRange() - can be used to implement the familiar tape recorder-style functions - play, stop, pause, resume, skip, fast forward, and rewind - as well as more advanced features needed in sound editors such as playing a highlighted segment.

Another member function is PlayPrepare(), which performs time consuming playback initialization, such

as paging in the first few seconds of a sound off of the disk. Calling PlayPrepare() before Play() guarantees that when Play() is eventually called, playback will commence with minimal latency.

Wait() causes a process to block until playback is finished. GetClock() returns a TClock that is synchronized to the sound's playback. This clock's offset is zero when the sound starts playing. The clock advances at the same rate as sound playback. This clock can be used for synchronizing events to playback of the sound.

Time Units

It is often necessary to indicate a position in a stream of audio. For example, GetPosition() has to return the value of the playback pointer. TTime objects are used to represent time in all parameters and return values in the Pink Sound Tools. TTime is subclassed into various time units, such as seconds and milliseconds. For programmers who would rather think in terms of sound samples instead of some other time unit, a subclass of TTime, TSoundSamples, is defined. TSoundSamples requires two parameters in it's constructor, a sample offset and a sample rate. These two together can be used to convert a TSoundSamples object to any other TTime object or subclass.

Relative Playback

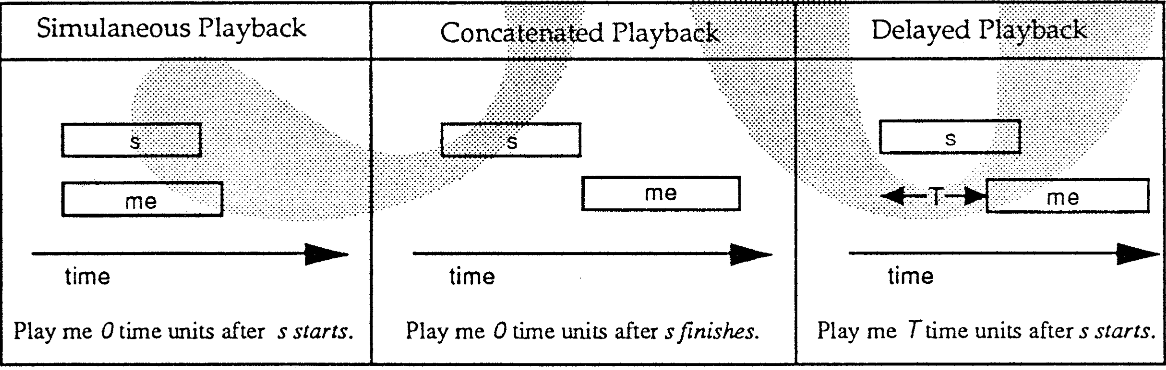
Often it is desirable to play two objects simultaneously. When playing the notes of a musical chord, for example, all notes should be started at once.

At other times one wants one sound to immediately follow another. Concatenating digitized phrases together for voice mail prompts is an example. When playing "You have five messages in your inbasket", one might play "You have" followed by "five" followed by "messages in your" followed by "inbasket".

Both of these cases can be handled by one general rule which can be applied to any audio player:

Play me <T> time units after audio player <S> <starts | finishes> playback.

Examples:



MAudioPlayer defines a member function, PlayWhen(), to handle this:

```
enum RelativeTo { kStart, kEnd };

PlayWhen(MAudioPlayer& s, TTime t,  RelativeTo startOrFinish);
```

Audio Player/Recorders

An audio player/recorder is an audio player which can also record audio. The C++ class

MAudioPlayerRecorder represents an audio player/recorder. It descends from MAudioPlayer. TSimpleSound, which plays and records sound files, is an example of an MAudioPlayerRecorder. MAudioPlayerRecorder has a Record() method to initiate recording. Recording normally starts at the beginning of the sound. SetRecordRange can be used to selectively record into a portion of the sound. RecordPrepare() performs time consuming preparation prior to recording, if any.

Recording normally causes audio to be inserted into the existing sound. ReplaceWhenRecording () can be called to cause audio data to be recorded over instead of inserted. Audio player/recorders that support multiple channels of sound will often use the replace feature, as sound needs to be recorded into one channel while maintaining synchrony with the other channels. InsertWhenRecording()) resets recording back to inserting instead of replacing.

System Objects

System objects are used to represent system resources such as the computer's speaker and microphone. TSystemSpeaker and TSystemMicrophone are objects that represent the computer's audio inputs and outputs. Each has a gain control. It is possible to connect processors, such a TReverb, into TSystemSpeaker. Doing this would cause all sounds coming from the computer to be reverberated. It is also possible to redirect audio within TSystemSpeaker and TMicrophone so that a telephone handset is used for input and output instead of the computer speaker and microphone.

Control-panel type applications can use these objects to implement system-wide volume controls.

A parenthetical note about system volume control: An end user must be able to instantly adjust the speaker volume. Imagine starting playback of a sensitive voice message. If the playback were too loud, one would instantly want to lower or even mute the volume. Spending ten seconds to launch a control panel application would be unacceptable. The best solution is to provide a physical knob that the user can twiddle. Another solution is to provide keyboard commands for raising, lowering, and muting speaker volume. These keyboard commands would have to work across all applications, implying that they would have to be intercepted by the system prior to being sent to an application. A third solution would be provide a graphical volume control that is always visible on the desktop. User interfaces for volume control will have to be investigated further.

Future Directions

There are a number of features that will be added to audio objects in releases of Pink beyond release 1.0. These are part of the architecture, and are discussed in the following sections.

Degradation

When there isn't enough CPU time to process all of the audio objects on a system, we need to degrade gracefully. There are a number of schemes for doing this. Each will be discussed here.

Least Recently Used Algorithm

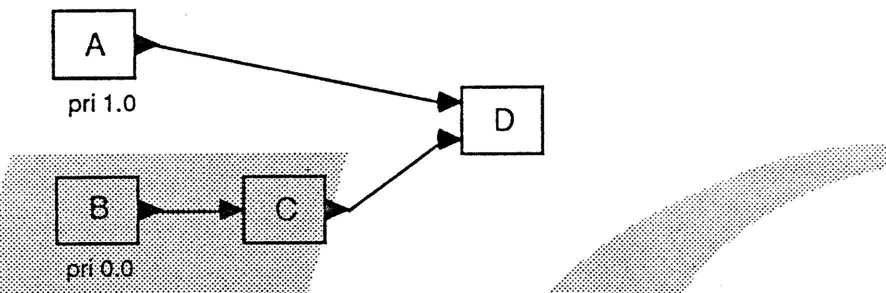
A method used in music synthesizers is to stop playing the oldest note when there are more notes than the system can handle. This works well because the oldest note has usually decayed the most in amplitude and can be dropped without notice.

In Pink, we could drop the oldest sound when we run out of CPU time. A "sound" in our case would be a playing audio player. Precedence would have to be given to audio player/recorders which are recording, as interrupting recording is generally more offensive than interrupting playback.

Priorities

With this scheme, each audio object has a member function to set and get a priority. Priorities are floating point values that range from 0 (lowest priority) to 1.0 (highest priority). When the system runs out of real-time, it will drop objects with lower priorities until real-time is reestablished. All objects at the same priority are dropped at once, so the developer can group objects together that should be dropped together. As real time becomes available, audio objects are added back in in reverse order.

One problem with this scheme is determining what to do with dangling audio inputs and outputs to objects as they are dropped. One solution is to only attach priorities only audio *producers*, such as audio players and microphone objects. As these objects are dropped, all objects that they feed would be dropped too, unless they were still being fed by some other object.



In the above example, audio objects B and C would be dropped first. B is dropped because it has the lowest priority. C is dropped because it depends entirely on B. D is not dropped because it still depends on A. If there was no real-time left, A and D would be dropped also.

Delayed Playback

Another degradation method would be to delay playback on those objects that could withstand it. Playing a sound to tell the user that the printer is out of paper, for example, could be delayed several seconds or so without causing harm. Delayed playback could be implemented by giving every audio player a SetTolerableDelay() call that would specify the maximum delay that could be tolerated. The default would be zero tolerable delay.

Degradation Levels

An enhancement to the priority scheme would be to add degradation levels to each object. With this method, the system, when it is running out of real time, can tell objects to perform in a degraded but still acceptable manner that uses less real-time. An eight voice polyphonic synthesizer, for example, could degrade to seven voices, six voices, all the way down to one voice. The most severe degradation level would be to drop the object all together.

Non Real-Time Operation

Normally, audio objects are run in real-time or, if the system doesn't have enough CPU time, they are dropped. Sometimes it is desirable to run objects in non-real-time. An example would be non-real-time signal processing, where complicated sounds are synthesized and stored on disk instead of played through a speaker. Member functions can be added to MAudio to set and clear a real time flag. When in non-real time mode, the object is executed at a low priority in the background. It is never dropped.

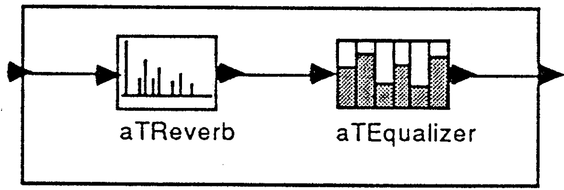
Smart Connections

A future enhancement would be to add a new function, TAudioPort::SmartConnect(), which will automatically insert a converter for the caller if the audio data types don't match. This form of weak type-checking could make programming easier, but it hides computationally expensive converters from the

programmer, which may not be ideal for programmers interested in predicting real-time response.

Audio Groups

Sometimes it is desirable to create audio objects that are composed of other audio objects. In the illustration below, a new instance of an audio object, anAveryFisherHall, has been created using instances of TReverb and TEqualizer.



anAveryFisherHall

anAveryFisherHall is an *audio group*. The instances of TReverb and TEqualizer are *components* of TAveryFisherHall. Audio groups are represented by the C++ class TAudioGroup, which descends from MAudio.

An audio group has a list of component MAudios. To create an audio group, component MAudios are added to the group's component list, new input and output ports are created, and the ports are connected together.

Control of component objects is accomplished by directly calling the component's member functions.

Play Groups

Often it is desirable to create a new, playable sound from component sounds. For example, one might combine eight sounds, each representing a note in a scale, into a new sound called a aScale. Or one might want to combine the sounds "You have", "5", and "messages" into a new sound called numMessages.

A *play group* is a group of audio objects containing at least one audio player. The play group, represented by the C++ class TAudioPlayGroup, is both an audio group and an audio player. It descends from both TAudioGroup and MAudioPlayer. It inherits MAudioPlayer's functions Play(), Stop(), PlayWhen() etc. It also has a member function SetFirstToPlay(), which tells the play group which MAudioPlayer to play first when the play group's Play() member function is called.

Using TAudioPlayGroup, complicated sounds can be created from component sounds. The relative start times of these sounds can be specified by their respective PlayWhen() functions. The play group's Play() method can then be called to fire off the whole shebang.

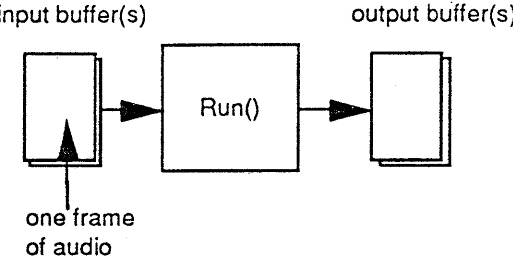
Writing New Audio Objects

It is expected that most users of the Pink Sound Tools will use existing audio objects, and not create their own. Creating new audio objects involves a specialized knowledge of sound and signal processing. This section is for those readers who would like to create their own audio objects.

Processing Audio

Connections between audio objects are implemented using buffers of audio data. Each connection, or "patchcord" has a buffer associated with it. Every audio object has a Run() member function which reads data from its input buffers, processes it, and writes the result to its output buffers. The size of the

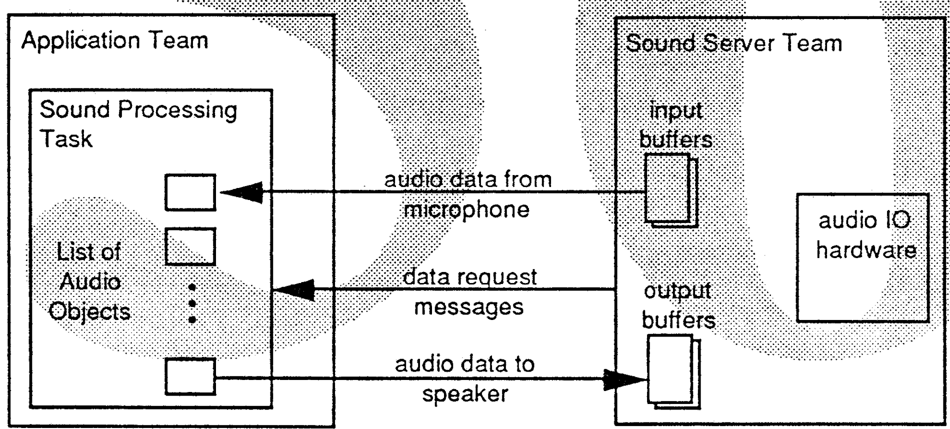
buffers is variable but small, from 1 to 1000 milliseconds worth of data. This amount of time is called a *frame.*



An audio object is either *active* or *inactive.* Active objects are producing or processing sound. Inactive objects are silent. A sound file player, for example, is inactive until it is told to play, at which point it becomes active. It remains active until playback stops, at which point it becomes inactive again.

Within each team there is a *sound processing task.* This task keeps a list of all of the audio objects in its team. The order of the list is determined by the consumer/producer relationships between the objects (producers must run before consumers). To process a frame's worth of audio data, the sound processing task steps through the list, calling the Run() function of each active object. The sound processing task must run in real-time, hence it runs at a high priority.

A *sound server* team takes the audio data produced by the various sound processing tasks, mixes the audio together, and plays them out on the computer's speaker. All sounds playing concurrently are heard simultaneously. When recording, the reverse happens. The sound server takes audio data coming in from the computer's microphone and hands it to those sound processing tasks that have microphone objects in them. The sound server owns the computer's sound IO hardware.



The sound server has internal buffers for input and output audio. These buffers are shared so that they can be accessed by the sound processing task. The output buffers are used for playback. They are filled by the sound processing tasks and emptied by the hardware. The sound server sends a message to the sound processing tasks requesting more data when the output buffers are nearing empty. Recording works in reverse. The sound server sends a message to the sound processing tasks requesting that they consume more data when the input buffers are nearing full.

Interface for Writing Audio Objects

MAudio has public and protected member functions which are used specifically for processing sound. These member functions are only used by those who wish to modify existing or write new audio objects. The member functions are divided into two categories, 1) signal processing and 2) activating and deacti-

vating.

Signal Processing

The Run() member function performs the actual signal processing. For a sound file player, that means getting audio from the sound file and writing it to the output buffer. For a reverberator it means taking samples from the input buffer, processing them to add reverb, and writing the result to the output buffer. Run() is almost always overridden when writing a new audio object. Because it is processing anywhere from 22,254 to 48,000 samples a second, Run() should be coded to execute as efficiently as possible. Run() can call protected member functions GetInputBuffers() and GetOutputBuffers() to get its input and output buffer lists, respectively.

A RunPrepare() member function can be overridden to perform time consuming preparation prior the Run(). Typically this is used for caching pointers to input and output buffers so Run() doesn't always have to spend time figuring out where its buffers are. RunPrepare() is called much less frequently than Run() - it is only called when an object first becomes active and whenever the number of it's input and output buffers change.

Objects that implement speakers can use the protected member function OutputAudio () to hand processed data to the sound server for playback. Likewise, microphone objects can use the protected member function InputAudio() to get input data from the sound server.

Activating and Deactivating

Most audio objects need not worry about activating or deactivating, as the MAudio takes care of this for them. There are some member functions of MAudio that can be of interest to certain objects, however. These are described below:

Activate() is called when an object first becomes active. This can be overridden if an object needs perform some initialization prior to activation. Filtering and reverberation objects might want to initialize filter delays, for example.

Deactivate () is called when an object is deactivated. If the object needs to clean up in some way, it can override this function.

Protected member function ActivateMe() must be called by an audio object to tell the sound processing task that it wants to become active. The sound processing task will call the audio object's Activate() function as soon as it finishes processing its current frame. The sound processing task needs to know when objects become active so it can activate the other objects downstream from it in the signal processing chain. ActivateMe() would be called when an MAudio is told to play. Normally, however, an audio object that can play would descend from MAudioPlayer and would not have to worry about calling ActivateMe() directly.

Protected member function DeactivateMe() is called when a server object wants to deactivate. For example, a MAudioPlayer would call this when it has finished playing a sound.

Hardware Independence

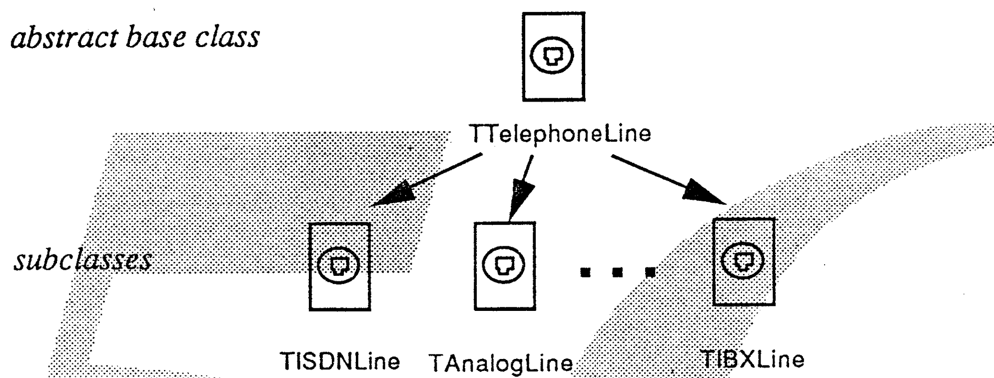
Hardware independence is achieved by providing new implementations of existing audio objects. Often, it is desirable to *extend* an object as well as create a new implementation. Extension is accomplished by subclassing audio objects. Let's illustrate this with an example.

There are many different types of telephone systems in use today - analog, Integrated Digital Services Network (ISDN), and private branch exchanges (PBXs) such as Rolm, Northern Telecom, AT&T, and InteCom. None of these systems are hardware compatible with each other, so all will require different

hardware interfaces to our personal computers.

Our goal is to provide a stable, standard programmer interface for dealing with telephones - regardless of the particular telephone system. We do this by defining an object that is an *abstraction* for the phone system, TTelephoneLine. We define member functions common to all phone systems, such as a means to make a call and answer the phone.

Now, let's say a developer wants to create an object for a particular phone system, lets say ISDN. The developer creates a subclass of TTelephoneLine, say TISDNLine. The developer overrides the functions defined in TTelephoneLine so that the implementation uses the ISDN hardware. Existing telephone applications defined using TTelephoneLine will work with the new TISDNLine object.

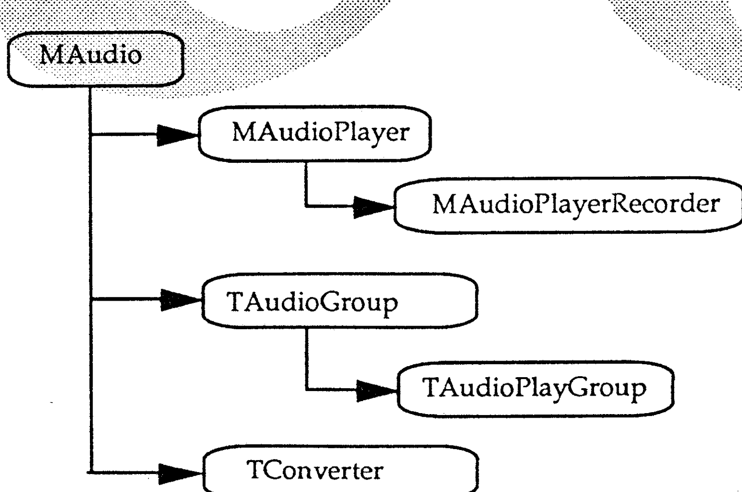


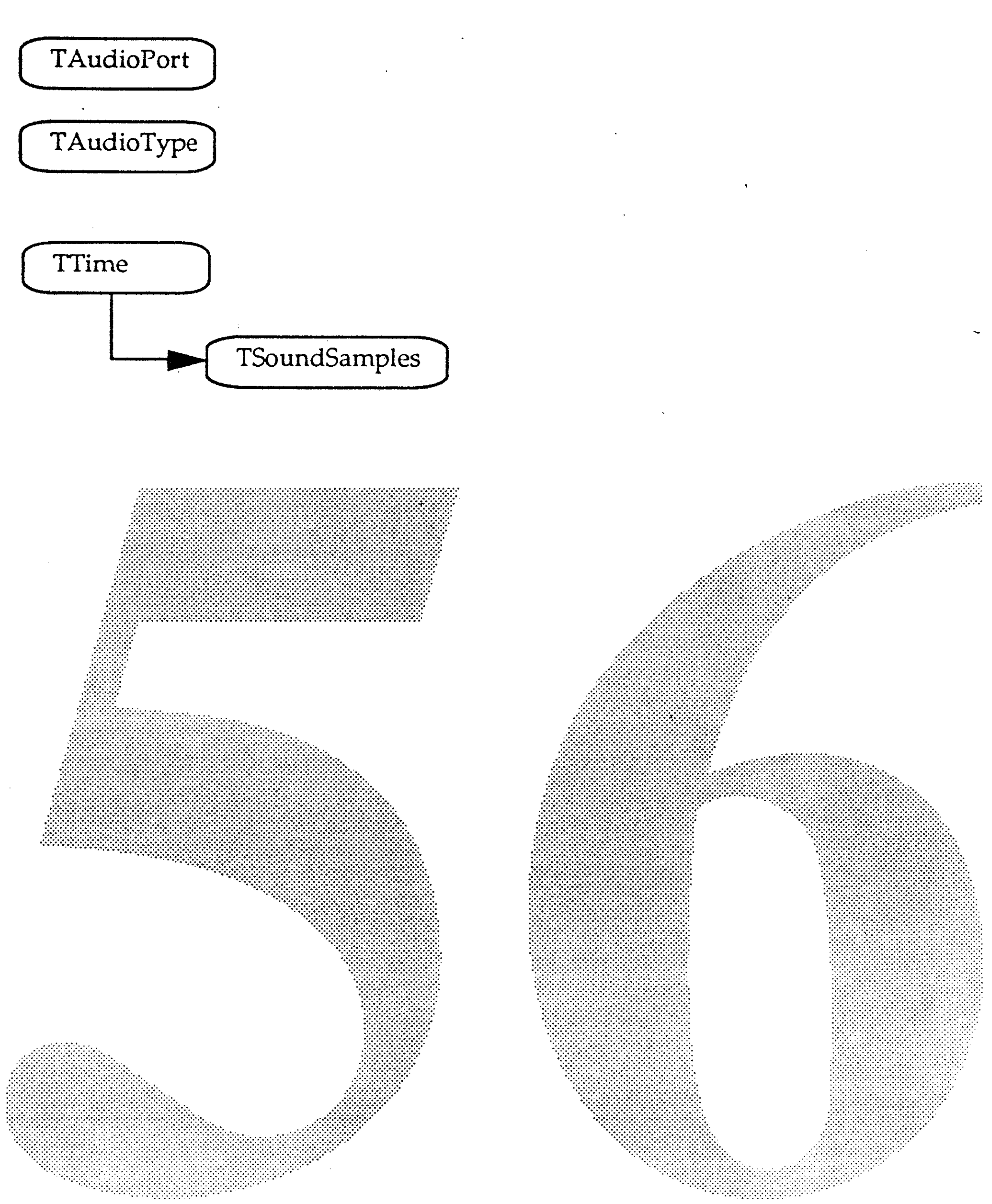
The developer can also extend TISDNLine by adding new functions to it. New applications can choose to take advantage of the new functionality provided by TISDNLine if they choose.

The subclasses are dynamically linked in with an application at runtime. That way, applications don't have to be recompiled to use a new implementation of an object.

Class Hierarchy

Note: All objects descend from MCollectible.




TAudioPort
TAudioType
TTime
TSoundSamples
56

Sound, Speech, & Telephony

56

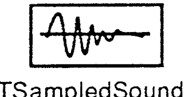
Introduction

An important feature of the audio objects are that they provide a stable, standard application programmer interface for audio, speech, telephony, and music applications. The Pink Sound Tools define a set audio objects to do the most common sound and speech operations. The definition of the public member functions of these classes is the application programmer interface. This section describes those classes.

Audio

This section describes classes for playing, recording, and processing sounds. Unless otherwise noted, all of these objects descend directly from MAudio.

TSampledSound


TSampledSound

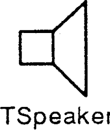
TSampledSound is used for playing and recording sound. It is an abstraction for a sound file. It has a TSpeaker and TMicrophone built into it, so no connections with other objects are necessary. It has no inputs and no outputs. It descends from MAudioPlayerRecorder. It inherits Play(), Stop(), and Record() methods as well as SetPlayRange() for playing segments of the sound, PlayWhen() for relative playing, Wait() for waiting for playback to finish and GetClock () for getting a clock so that events and alarms can be synchronized to playback and recording. It also has Delete(), Insert(), Copy() and Replace() functions for editting the sound file. It has GetVolume() and SetVolume() calls for changing the output volume.

The main design goal of TSampledSound is that it be simple to use. Playing a sound is as simple as:

```
TSampledSound    s("/Sounds/Beep");
s.Play ();
```

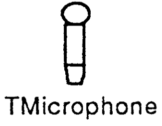
Because of it's simplicity, TSampledSound is the class of choice for most programmers who want to add sound playback to their applications.

TSpeaker


TSpeaker

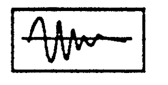
TSpeaker plays audio data on the computer speaker. It can have one or two inputs (mono or stereo). It has a SetVolume() function for each channel. Many instances of TSpeaker can exist, either in one task or in different tasks and teams. The output of each TSpeaker is summed and played out on the system speaker.

TMicrophone


TMicrophone

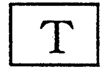
TMicrophone records audio from a microphone or line input. Both built-in microphones (a la Elsie) or external microphones (Farallon Macrecorder) can be accommodated. TMicrophone has one output.

TSimpleSound


TSimpleSound

TSimpleSound is identical to TSampledSound except that it does not have a built in speaker and microphone. Instead, it has one output and one input which can be connected to any other audio object.

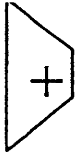
TConverter



TConverter

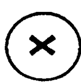
TConverter converts its input to another audio type and outputs the same audio data but in a different format. It has one input and one output. The constructor for TConverter is passed parameters to specify what conversion should take place.

TMixer


TMixer

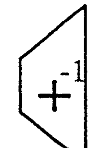
TMixer is a very basic mixer that will sum two or more inputs into a single output. More complicated mixers that sum to stereo and provide gain and panning on each input can be constructed from TMixer or hardcoded later.

TGain


TGain

TGain can increase or decrease the gain of a signal. It has a single input and a single output. Large signal levels will be clipped. It has SetGain () function.

TSplitter



TSplitter

TSplitter takes one input and splits it into one or more outputs.

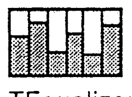
Audio Processing

TReverb

TReverb

TReverb causes its input to sound like it is in a large room or hall. It adds a pleasing effect to music and sound effects. TReverb has one input and one output. It has a SetMix() function to determine the ratio of processed to unprocessed sound. It has a SetRoomSize() function to set the size of the room being emulated.

TEqualizer


TEqualizer

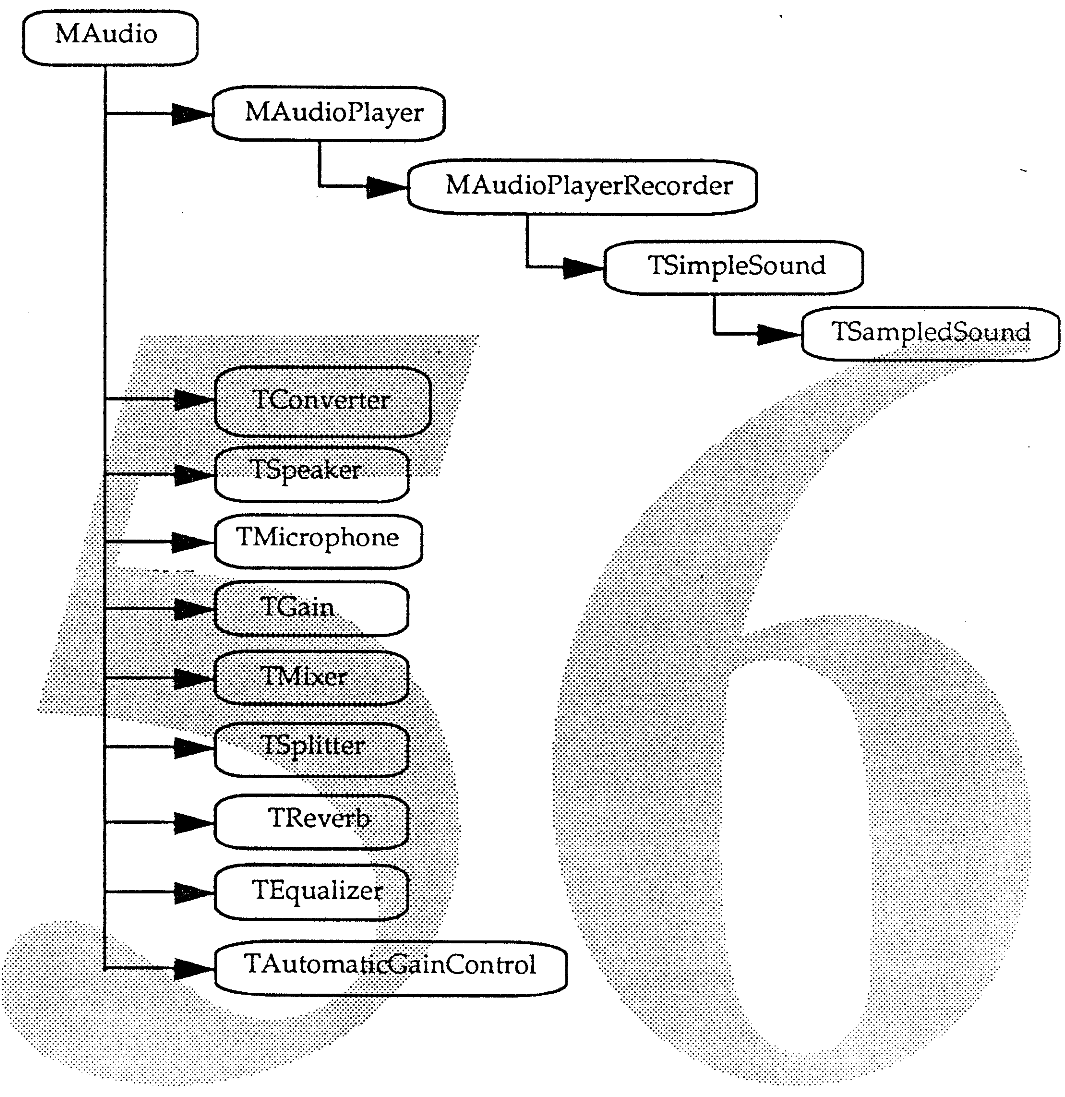
TEqualizer amplifies and attenuates various frequencies in the input signal. It has one audio input and one audio output. It has functions to set the levels of the various frequency bands.

TAutomaticGainControl

TAutoGainControl

TAutomaticGainControl implements an automatic gain control. It has one input and one output. It is used to keep volume levels constant when recording. TAutoGainControl is especially useful when recording voice from a telephone line

Class Hierarchy



Speech

This section describes classes for speech processing.

Text-to-Speech Synthesis

A Text-to-Speech synthesizer converts text into speech. There are a variety of classes associated with this process, the simplest being TTextToSpeechSynthesizer.

TTextToSpeechSynthesizer

TTextToSpeech-
Synthesizer

TTextToSpeechSynthesizer is an abstraction for a text-to-speech synthesizer. It inherits from MAudioPlayer. It has one audio output. It has a SetInput() function that takes the text which is to be converted to speech. When Play() is called, this text is spoken. It has other functions to set the voice in which it is going to speak. It can send events to a task when specified words have been spoken. It can use application specific pronunciation dictionaries to help it correctly pronounce words.

Converting Text-to-Speech

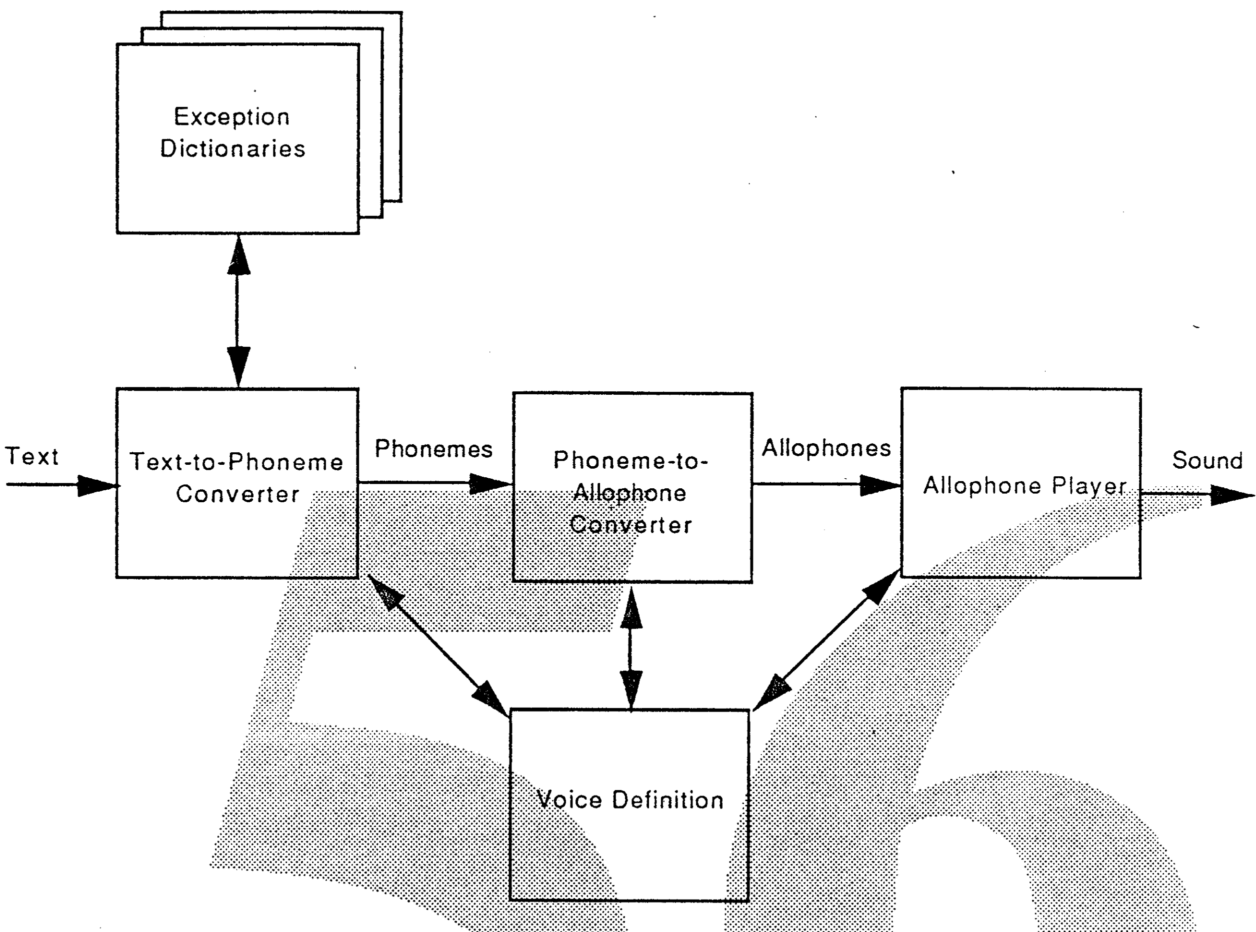
For programmers who want more control over the synthesis process, additional classes are available. There are three distinct phases in the process of converting arbitrary ASCII text into spoken words, phrases, and sentences. The first phase involves deciphering language-specific spelling conventions and generating an unambiguous representation of the speech sounds, or *phonemes*, required to pronounce a given stream of text. Conversion of text to phonemes is accomplished by means of a set of spelling rules and a built-in dictionary of exceptions to these rules, which can be augmented by the addition of one or more custom dictionaries embodying pronunciations preferred by an individual user or appropriate for a specific application.

The second phase in the process of text-to-speech conversion replaces the relatively abstract phonemic representation with an explicit description of how speech sounds corresponding to the original text are to be realized in context. The output of this phase of processing is a list of the positional variants of the required phonemes, or *allophones*, along with control parameters specifying the pitch and duration of each item.

The final step consists of accessing pre-stored digital data corresponding to each pair of allophones, modifying their pitch and duration in accordance with the control parameters included in the allophone stream, concatenating the resulting sound segments into a buffer of continuous audio data, and playing the buffer as a sampled sound.

Pink Sound Tools will support the use of multiple synthetic voices. These voices will differ in such features as speaking rate, speaking style, baseline pitch, and pitch range, as well as in the language each voice is designed to speak and the actual digital data used to play out allophones.

The following diagram illustrates the relationships between these different elements.



The classes described below provide alternate entry and exit points at the beginning and end of each phase of processing. Each class is used internally by TTextToSpeechSynthesizer.

TTextToPhonemeConverter

TTextToPhonemeConverter is an abstraction for the first phase in the process of generating speech from arbitrary text. It is not an audio object. It has a SetInput() function which takes in text to be converted to phonemes. It has two output functions: GetVerboseOutput() produces a human-readable but redundant phoneme specification; GetTerseOutput() produces a compact, coded representation suitable for efficient storage. The class also has functions which set the characteristics of the synthesized voice which affect application of the text-to-phoneme conversion rules, and functions which permit addition and deletion of customized pronunciation dictionaries to augment the built-in exception dictionary. TTextToPhonemeConverter might be used to facilitate the process of creating a new custom dictionary, since it could be invoked to produce a first-pass transcription of problematic words which would then be edited and corrected.

TPhonemeToAllophoneConverter

TPhonemeToAllophoneConverter is an abstraction for the second phase in the process of generating speech from arbitrary text. It is not an audio object. It has a SetInput() function which takes in phonemes to be converted to allophones. It has two output functions: GetVerboseOutput() produces a human-readable but redundant allophone specification; GetTerseOutput() produces a compact, coded representation suitable for efficient storage. The class also has functions which set the characteristics of the synthesized voice which affect application of the phoneme-to-allophone conversion rules.

TAllophoneToSpeechSynthesizer

ToAllophoneToSpeechSynthesizer is an abstraction for the final phase in the process of generating speech from arbitrary text. It inherits from MAudioPlayer. It has one audio output. It has a SetInput() function which takes in allophones and their associated control parameters. When Play() is called, these allophones are spoken. The class also has functions which identify the data tables from which segments are to be extracted for concatenation and which specify other global voice characteristics which affect the synthesis process.

TPhonemeToSpeechSynthesizer

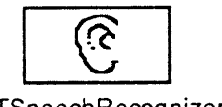
TPhonemeToSpeechSynthesizer is an abstraction which subsumes the second and third phases in the process of generating speech from arbitrary text. It inherits from MAudioPlayer. It has one audio output. It has a SetInput() function which takes in phonemes to be converted to speech. When Play() is called, these phonemes are spoken. The class also has functions which specify global voice characteristics which affect application of the phoneme-to-allophone conversion rules and the process of playing out allophones. TPhonemeToSpeechConverter would be used to avoid the overhead of text-to-phoneme conversion when large amounts of phonemic material unmixed with straight text are to be synthesized.

TVoiceDefinition

TVoiceDefinition is a class which contains a variety of information relating to the specific characteristics of the voice to be synthesized. It is not an audio object. Various member functions of the class are called at each stage in the process of converting text to speech. TVoiceDefinition has functions which permit modification of this information. SetSpeakingRate() specifies the average number of words to be spoken per minute. SetBaselinePitch() specifies the starting point for the fundamental frequency of the voice. SetPitchRange() specifies the absolute maximum and minimum values which the fundamental frequency of the voice may attain. SetSpeakingStyle() permits varying degrees of formality and articulatory precision to be requested. SetPauseFactor() changes the average length of pauses inserted between phrases and sentences without otherwise affecting the rate at which words are produced. SetLanguage() defines the rule sets to be used in generating phonemes and allophones to be spoken. (These will differ significantly from language to language.) Finally, SetSpeechTable() identifies the set of pre-stored segments to be used during the final stage of processing.

Speech Recognition

TSpeechRecognizer

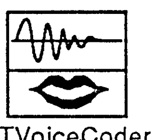


TSpeechRecognizer

TSpeechRecognizer is an abstraction for a speech recognizer. It inherits from TAudio. It has one audio input. It has Record() and Stop() functions to start and stop recognizing, respectively. It can send events when specific words or phrases have been recognized. There are likely to be many subclasses of TSpeechRecognizer, each tailored to specific speech algorithms. The definition of TSpeechRecognizer should be generic enough, however, that it can provide a stable interface for applications that use speech recognition.

Speech Recording and Playback

TVoiceCoder



TVoiceCoder

TVoiceCoder is a descendant of TSimpleSound. It has one input and one output. It overrides Record() so that when recording, it removes silence from the audio signal, thus saving disk space if you know the signal will only contain voice. It can change the rate of playback without changing pitch. It can store the voice in a highly compressed form suitable only for speech. This object is useful for phone answering and voice mail applications.

Class Hierarchy

- MAudio
  - MAudioPlayer
    - MAudioPlayerRecorder
      - TSimpleSound
        - TVoiceCoder
    - TTextToSpeechSynthesizer
    - TAllophoneToSpeechSynthesizer
    - TPhonemeToSpeechSynthesizer
  - TSpeechRecognizer

TTextToPhonemeConverter

TPhonemeToAllophoneConverter

TVoiceDefinition

Telephony

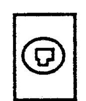
This section describes classes for building telephone-based applications. Unless otherwise noted, all of these objects descend directly from MAudio.

TTelephoneSet

TTelephoneSet

TTelephoneSet represents a local telephone attached to the computer. It can be recorded from and played to. Many instances of it can exist. It can send events to a task when the handset goes on and off hook and when buttons are pressed.

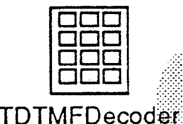
TTelephoneLine



TTelephoneLine

TTelephoneLine represents the phone line coming from the central office or PBX. It is an abstraction for the *voice* features of a phone line, but not the *data communications* features (such as those available with ISDN). Data communications features are handled by the Pink Networking tools. TTelephoneLine has one audio input and one audio output. It can be recorded from or played to. It can send a message to a task when the phone line rings. It has functions to answer the phone, make phone calls, and perform functions such as hold, conference, and transfer. TTelephoneLine would be used to implement a phone answering machine or remote access application.

TDTMFDecoder



TDTMFDecoder

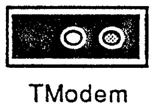
TDTMFDecoder is an abstract Dual Tone Multi Frequency (DTMF, commonly known as Touch Tones) detector. It has one audio input. When it recognizes a touch tone, it posts an event indicating which tone was depressed. It has Record() and Stop() functions to start and stop tone recognition, respectively.

TDTMFGenerator

TDTMFGenerator

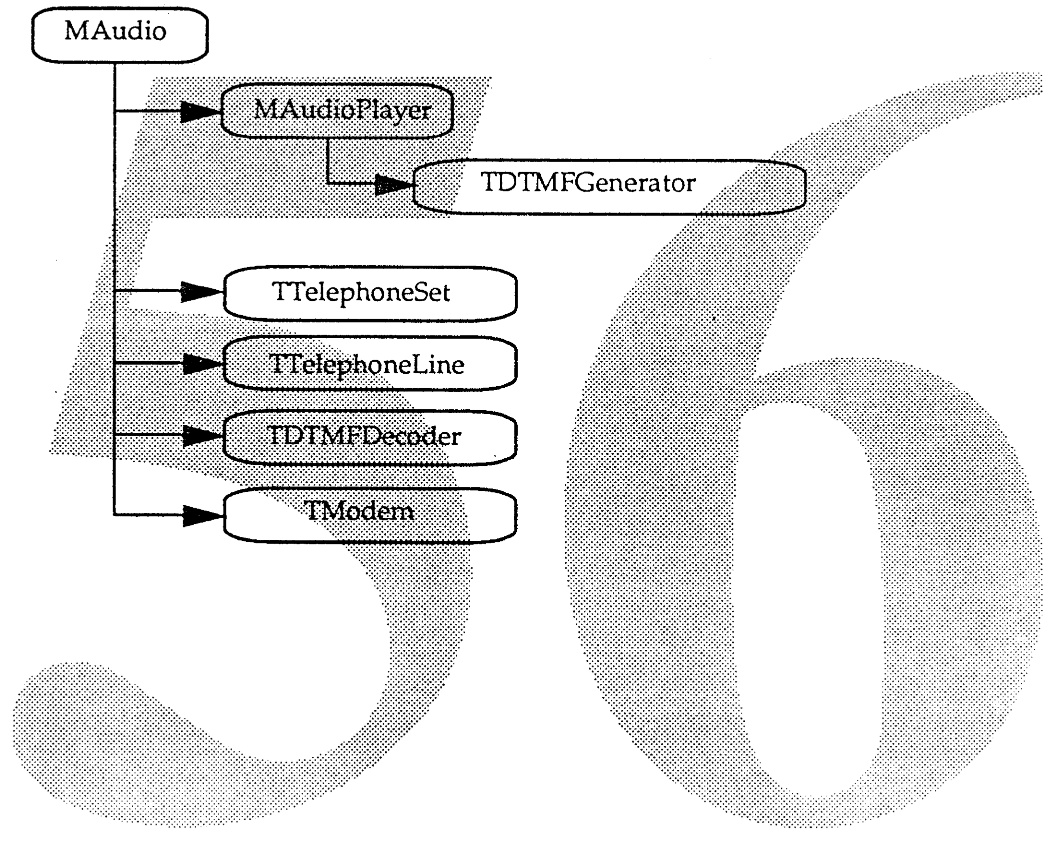
TToneGenerator generates Dual Tone Multi Frequency tones(DTMF, commonly known as Touch Tones). It descends from MAudioPlayer. It has one audio output.

TModem



TModem converts audio data into digital data. It has one audio input and one audio output. It has functions for setting equalization, call progress monitoring, and configuration. TModem must be designed in conjunction with networking to see how it fits into their world.

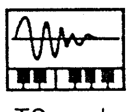
Class Hierarchy



Music

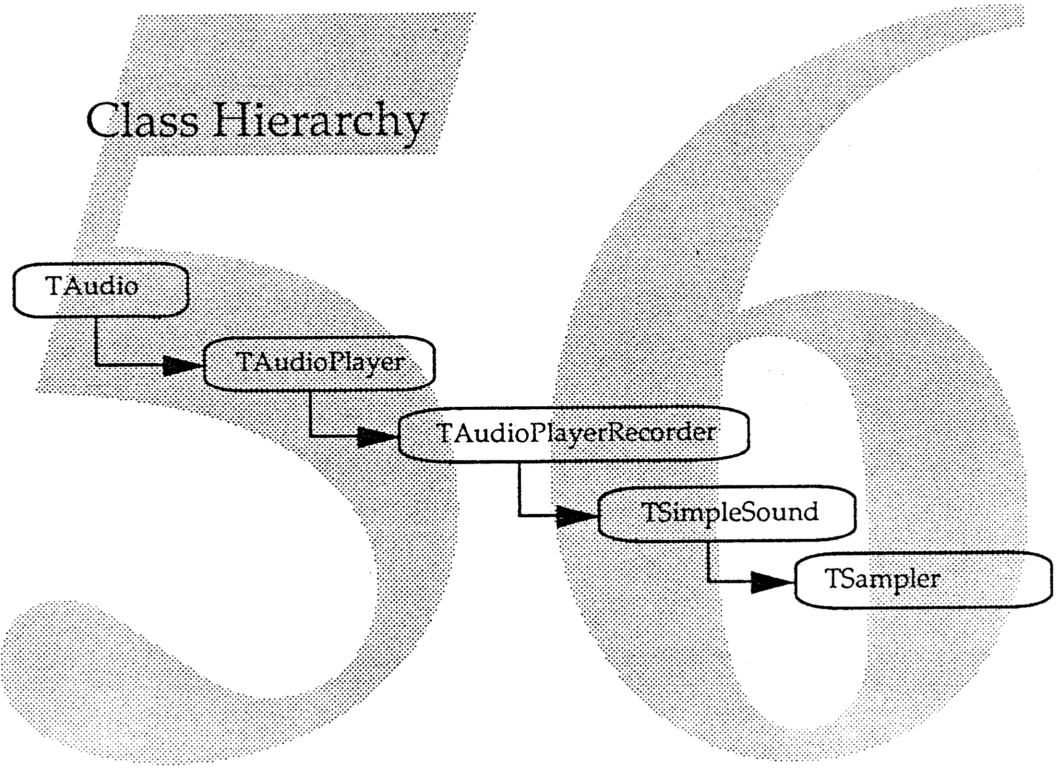
This section describes classes that can be used for music synthesis. Of course, all of the objects found in the Audio chapter can be used for musical purposes too.

TSampler



TSampler

TSampler is a descendant of TSimpleSound and inherits all of its functions. It has new features that make it useful for music, however. It has one (mono) or two (stereo) outputs and one or two inputs. It can play sounds at an arbitrary pitch.



56

Editors

56

Editors

The Pink Sound Tools will provide editors for editing sounds and graphically connecting audio objects. All of these editors are built from objects that any application can create and use directly. The editors serve two purposes, 1) to give developers tools to create sounds and connections of audio objects for use in their programs, and 2) to provide a standard interface for end users for doing the same.

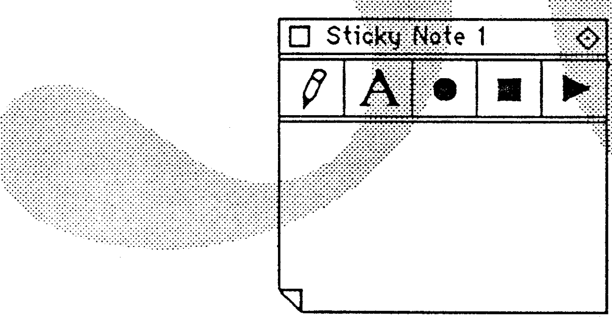
Sound Editor

A sound editor gives developers a means to create and edit sounds played by the audio object classes. It also provides a standard user interface for creating and editing sounds for voice annotation, voice mail, sounds for presentations, and the like.

The simplest way to record and play sounds is by using a sound palette. The sound palette resembles the controls on a standard tape recorder:

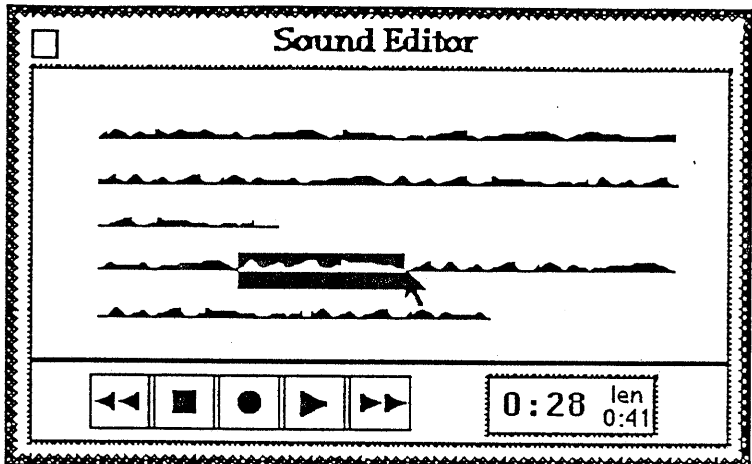
It has buttons for rewind, stop, record, play, and fast forward. These buttons behave in a manner consistent with a physical tape recorder (although to record you won't have to hold down the play and record buttons at the same time, just hit the record button). TSoundPalette is a C++ object that implements the sound palette. TSoundPalette has methods for creating a sound, saving the sound in a sound file, reading in a new sound, and functions to handle mouse hits on the tape recorder keys.

A subset of the sound palette, using only record, stop, and play keys, can be used by an application. In the following example, a "sticky note" used for voice, text, and graphical annotation, uses the sound palette subset.



When the fast forward and rewind buttons are eliminated, the play button always plays from the beginning of the sound, and the record button deletes the existing sound and makes a fresh recording.

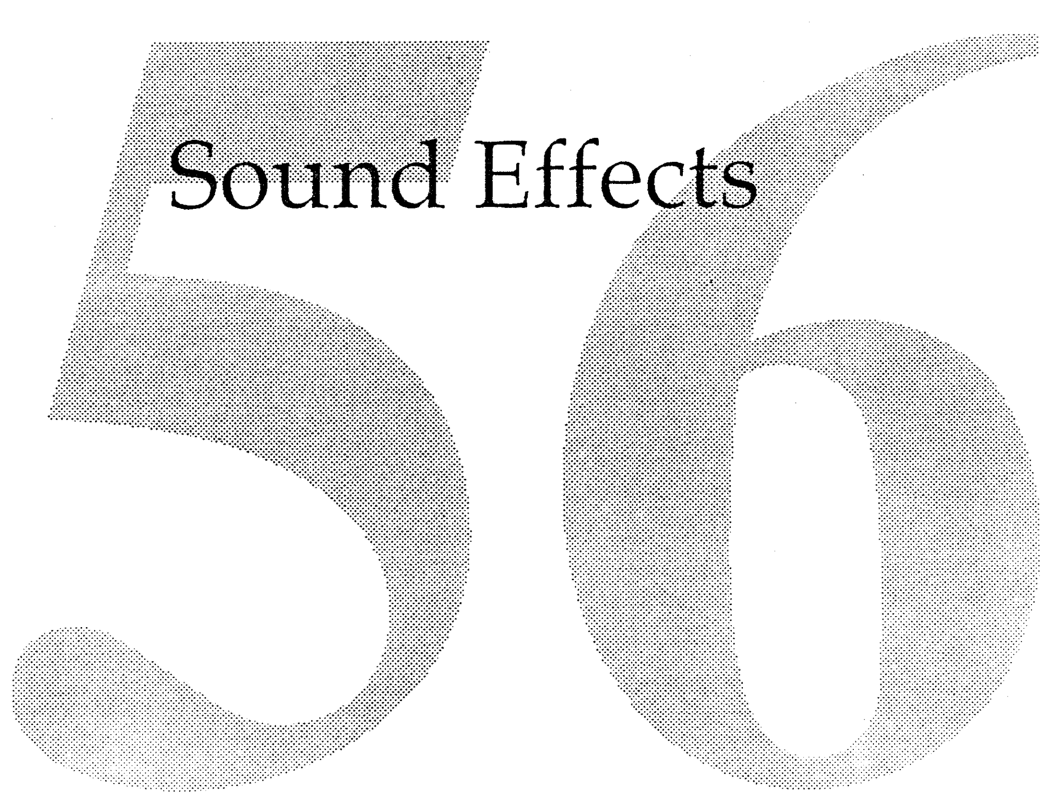
TSoundEdit is a C++ object which displays a visual representation of the sound. This object would always be used in conjunction with a TSoundPalette object to give the user visual control over editing the sound. Functions are provided to follow the mouse for dragging and selecting, query the view for the selection bounds, change the size of the view, scroll, and zoom in and out. Applications are encouraged to use TSoundEdit and TSoundPalette together if they require precision editing of the sound. We will provide a sound editor application, based upon TSoundEdit and TSoundPalette, which developers and users alike can use to create and edit sounds. The editor, in a prototype Pink window, is shown in this illustration:



It is expected that third party developers can make a business out of providing more powerful subclasses of TSoundEdit and TSoundPalette as well as more capable and specialized sound editors.

Audio Object Editor

Connecting audio objects together is inherently a visual process. In the section on audio objects, we outlined the C++ syntax for doing this, which is inherently non-visual. The audio object editor would allow developers to create connections of audio objects using a visual editor. Each object would have an icon associated with it. These icons could be chosen from a palette an connected together by dragging and rubber-banding lines between two icons. Potentially, the audio icons themselves could be edited by double clicking on them. This would display the source code for the audio object. It is possible that we might in the future use a visual digital signal processing language to create the audio icons.



Sound Effects

56

Sound Effects Library

Sounds are difficult to create from scratch. It takes a great deal of experience, knowledge, and patience to record or synthesize sounds. Recording sounds from movies, television, and even sound effects records can violate U.S. and international copyright laws.

We want all developers, most of whom are unfamiliar with sound, to be able to add sound feedback to their programs. End users will want sounds to use in sound track presentations or dramatic voice messages. The Pink Sound Tools will include a sound effects library to be used by both developers and end users. The sounds can be used in conjunction with the Pink Sound Tools without violating any copyright laws.

The size of the library need not be large. Anywhere from 50 to 200 sounds should be sufficient. The point of providing the library is to give developers and end-users sounds to start playing with. Third party developers can extend the library. It should be possible for a third party developer to make a business out of providing sound effects to add the the library.

56

Base Text Classes

Text Storage & Style Managment

56

Base Text Classes

Version for Big Pink Three

Roger Webster
x48115

This ERS outlines the basic text classes used for the manipulation of text under Pink. The collection of classes supports various types of text, selections, international characters, and conversion from eight to sixteen bits and back.

The text classes in this document do not support rendering or measuring text; they are for manipulating text in memory only.

56

Introduction

The classes described in this document manipulate character and style data, and are the basis for the text layout classes (please refer to the *ZZText* ERS). The classes described in this paper represent the lowest level string manipulation available under Pink, replacing functionality such as *strlen* and *strcat*. Even the concept of "length" is maintained in an implementation-specific manner (*length*, for example, is defined to be the number of characters, or sixteen bit entities in the string, not the number of bytes).

Note that the root of the text hierarchy, *TBaseText*, has no fields and a protected constructor. The *TText* class implements the protocol defined in *TBaseText*. For the purpose of maintaining the actual character data, an abstract class *TTextStorage* is defined, supporting insertion and deletion of characters from a stream of text, as well as accessing characters and substrings. At present there are two descendants of this base storage class: *TCheapTextStorage*, which implements an pointer to a block of data (possibly compressed), and *TFancyTextStorage*, which implements character storage as a recursive run array (for efficient insertions and deletions).

In addition to the base classes, there are several style-related classes: *TStyle, TStyleSet*, and *MStyleRuns*. *TStyle* implements a name-value pair for a particular style, such as "italic." A style set is a collection of styles that apply to a run of text. Each node in a style run is a style set and a count (maintained in the run array structures) of the number of characters across which the particular style set applies. The classes *MStyleRuns* and *TText* are combined to produce the lower-common denominator for accessing and maintaining styled text. The resulting class is *TStyledText*.

One goal of this design is to provide a model that supports the benefits of a sixteen-bit character set without all of the obvious cost. One feature of these classes (eventually) will be compression into fewer bits, probably eight, whenever possible. Note that this is not currently implemented. It is worthwhile to note that if the text being represented is a random collection of characters with no common language, script, or other restricting attribute that the overhead for this is a one-node run array that identifies the run as sixteen-bit data. A goal of the implementation is to minimize the storage required *regardless* of the composition of the text being stored, while also minimizing access times to textual elements (e.g., characters, substrings, styles, etc.).

For more information about higher-level facilities, please refer to the *Line Layout Manager* ERS and the *ZZText ERS*.

Theory Of Operation

The base text classes implement styled and unstyled text in a way that attempts to choose a storage scheme that is appropriate for the manner in which the text is being used. The decision about which scheme is appropriate is made dynamically, and may change many times during the life of a text object.

Two classes of particular interest within the base text hierarchy are *TText* and *TStyledText*. TText and its descendants, such as TStyledText, toggle between using an array of (currently uncompressed) sixteen-bit characters and a recursive run array of arrays of sixteen-bit characters, depending on the length of the string. For strings less than a certtain number of bytes long, the "flat" string structure is used. When a string reaches the maximum flat structure size bytes, it is split into "runs" of data thre-quarters of the maximum number of bytes per flat structure (allowing for some insertions into nodes thus created; the pain is incurred when the transmogrification occurs). When a string shrinks below the three-quarter limit, it is automatically "flattened" out. The reason there is a gap in the numbers is to prevent the string from "breathing," or flattening and expanding constantly if only a few characters are inserted and then deleted.

The theory behind the design of the string classes is to design a manipulation text model that allows clients to design their own mix-in classes, as *MStyleRuns* is a mix-in, and create a new derived class. For example, supporting a hypertext class might be accomplished by designing the *MHyperText* mix-in and creating a new text class, *TStyledHyperText*, derived from *TStyledText* (itself derived from *TText* and *MStyleRuns*), and *MHyperText*. In this scenario, the resulting hypertext class would be notified of any text insertions and deletions just as the style information is informed in the current implementation (via the *InsertAt* and *DeleteAt* methods). The only extra pain the client programmer must incur is be to make sure these insertion and deletion methods are called from the newly derived class.

Another goal is to support an unbounded set of styles, including user-defined styles. This will (hopefully) allow someone to create a special effect in some program such as LetraStudio and name the result, and then export the entire style specification to other programs. This would allow text not actually entered (i.e., typed in) within the scope of the program directly supporting the "special effects" to nonetheless make use of them, even for newly entered text. The style scheme is open: styles define their data. Only style names are implemented in the base protocol.

TBaseText

TBaseText is an abstract base class from which all of the text storage management classes are derived. It has no fields and its constructors are protected.

At present, this class supports the minimal text manipulation set of functions. It is our belief that any other functionality, such as string concatenation, can be achieved through a trivial combination of the methods provided. Please see the *Open Questions* section at the end of this document.

TText

TText is the general-purpose text class. It may be thought of as a dynamic array of sixteen-bit characters, though the internal representation of the text data varies depending on the amount of data and the patterns of usage. Text may or may not be compressed, and the compression decision will likely be an "intelligent" one that weighs the cost of compression against the cost of straight sixteen-bit encoding. In addition, the point at which the data representation changes from one contiguous block to multiple chunks of text will be determined by this class.

TTextSelection

This class implements a subrange of text. Many methods take an argument of class *TTextSelection*. An insertion point in the formatting classes is implemented as a *TTextSelection* with length is zero.

TStyle

This class implements a named property (a style) for text. The name may either be text object or a token. In the case of a text object, the name is first tokenized and then stored in the name field. As an example, consider a style named "bold." Somewhere on the system there will likely be a clearing house for standard styles names (and other items), so the token for bold should be well-known (this is under investigation).

N. B. The method *IsEqual* is provided, and returns true if the style names match. It is the reponsibility of the client to implement *IsSame* to do the right thing if two styles are equal.

TStyleSet

This class implements a named, expandable set of "styles" for a given run of text. In this context, even font changes are considered "styles." A client specifies a style as a TStyle object and may add or remove it from the set of styles managed by this class. Query methods are also provided to determine the contents of a style set.

Note that style sets are reference counted. Style sets are often transient entities in style runs, and to protect them from being deleted automatically by methods in the Utility Classes, all constructors have an *auto increment* parameter, which defaults to true (which means that the reference count field should be initialized to one). Clients that do not override the auto increment value and set it to false are responsible for deleting style sets themselves. Normally, style sets are created without initially incrementing their reference counts only by the text classes, and only for style sets that are generated dynamically as other runs are split. Note also that the styles that compose a style set are *not* deleted by the destructor of the style sets that contain them (this is deliberate—styles may exist in many style sets simultaneously).

As an example, a style set might consist of the styles (tokens and associated data) "bold," "italic," and "underline." Typical usage might be:

```
TStyleSet* aSet = new TStyleSet();
aSet->AddStyleToSet(boldStyle);
aSet->AddStyleToSet(italicStyle);
aSet->AddStyleToSet(underlineStyle);
```

The objects *boldStyle*, *italicStyle*, and *underlineStyle* are objects of class *TStyle* from above. They could be created in the following manner (note that this does not factor in the source of the name for the style *Bold*):

```
TToken boldStyleName(PartialChar *) "Bold");
TStyle* boldStyle = new TStyle(boldStyleName);

// Ditto for the other two styles...
```

MBaseRun

This class implements basic functionality needed to manipulate runs of information. It has one field of class *TRunArray* and provides methods to find the previous run, the next run, and the length of a run from an index. It also has two methods, *DoInsertBefore* and *DoDeleteAt* that are called to keep the runs in synch with other data.

MStyleRuns

This class implements runs of style sets. Each node in the run array is of class *TStyleSet*, the expandable style set from above. This class manages the maintenance of runs of style sets as text is added to or removed from the text with which a style run is associated (it is normally used as in *TStyledText*, see below).

The method *SetStyleInRange* is additive. That is, it performs a union of the styles in the set specified in the call with all style sets encountered in the range given. In the example below, if the run of [italic] is set in the middle of a long run of [bold], there will be a run of [bold] followed by a run of [bold, italic], followed by another run of [bold]. This is contrasted with another possible alternative: [bold] followed by [italic] followed by [bold]. Use *ReplaceStyleInRange* to use a new style set ragardless of any existing styles.

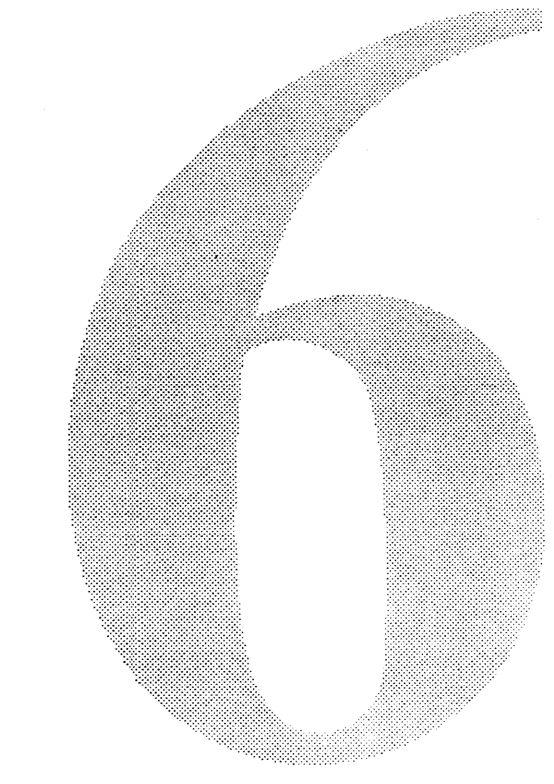
TStyledText

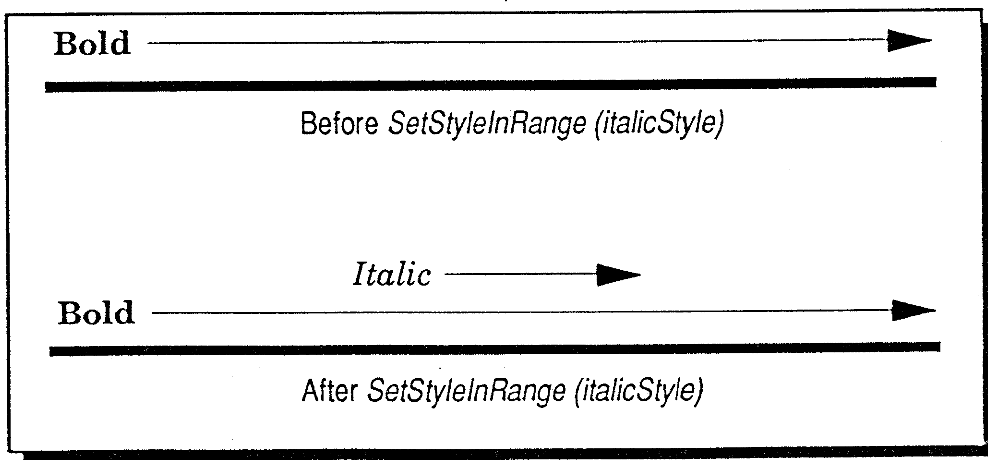
This class is a combination of TText and MStyleRuns. It provides a "styled text" class and is defined primarily for the purpose of preventing everyone from rolling their own. Note: aside from the methods that this class inherits it adds only constructors and destructors, plus two insertion and deletion notification methods. These last two routines are overrides of the ones specified in the base TBaseText class and simply call methods in MStyleRuns such as "DoDeleteAt" and "DoInsertBefore."

Note also that the recommended approach for clients to use when adding additional runs to a text object would be to provide a mixim such as MStyleRuns, say MVoodooRuns, and create a new derived class such as TVoodooText, or even TVoodooStyledText. The exact definition of MVoodooRuns is left as an exercise for the reader.

MSearchText

This class supports basic text searching on an object in the TBaseText hierarchy. There are a good many language-dependent features that have yet to be thrashed out (what does "case sensitive" mean, for example, and who will implement it), so only basic, hopefully fast, searching will be supported for now in this class. In addition, the careful reader will realize that the international group has yet to be involved in this; arguably a bug.





Examples

Creation of text objects is accomplished as follows:

```
TText* someText = new TText((PartialChar *) "Hi there.");
TText someMoreText = TText((PartialChar *) "Hello world.");
```

The nth character of a string can be referenced by:

```
UniChar c = someMoreText[n];
```

But note that...

```
someMoreText[n] = c;                // ...is not supported. Use InsertText instead.
```

Text from one text object may be inserted into another one:

```
someText->InsertText(someMoreText, 5, someMoreText->Length());
```

...which results in all of the text in someMoreText being inserted before the first "e" in "Hi there." in someText, yielding: *"Hi thHello world.ere."*

Data from a text object may be deleted as in the following:

```
someText->DeleteText(5, someMoreText->Length());
```

...which returns someText to its initial state.

Data may be extracted from a text object in any of several ways. The methods differ only in the type of data they return. In the first example, PartialChar* data is returned (essentially unsigned char*):

```
PartialChar* partialChars = (PartialChar *) malloc(100);
someText->ExtractText(0, someText->Length(), partialChars);
```

In the next example, UniChar* data is returned:

```
UniChar* uniChars = (UniChar *) malloc(200);
someText->ExtractText(0, someText->Length(), uniChars);
```

In the last example, a full text object is returned:

```
TText testText;
someText->ExtractText(0, someText->Length(), testText);
```

The assignment operator is overloaded, so that the following construct does the right thing:

```
testText = *someText;
```

Styled text is manipulated in exactly the same manner as unstyled text (as in the examples above). Styles are created as in the following sample code:

```
TPointSizeStyle pst(12.0);  // TPointSizeStyle is defined in "SystemStyles.h"
TStyleSet* ss1 = new TStyleSet();
```

To actually fill up a style set with styles, the client must call *AddStyleToSet* for each style to be included.

```
ss->AddStyleToSet(pst);
TStyledText* text = new TStyledText((PartialChar *) "Sample Text");
```

Selections simply maintain a pair of indices, and are used frequently to specify the extent over which a change such as a style run is to be applied.

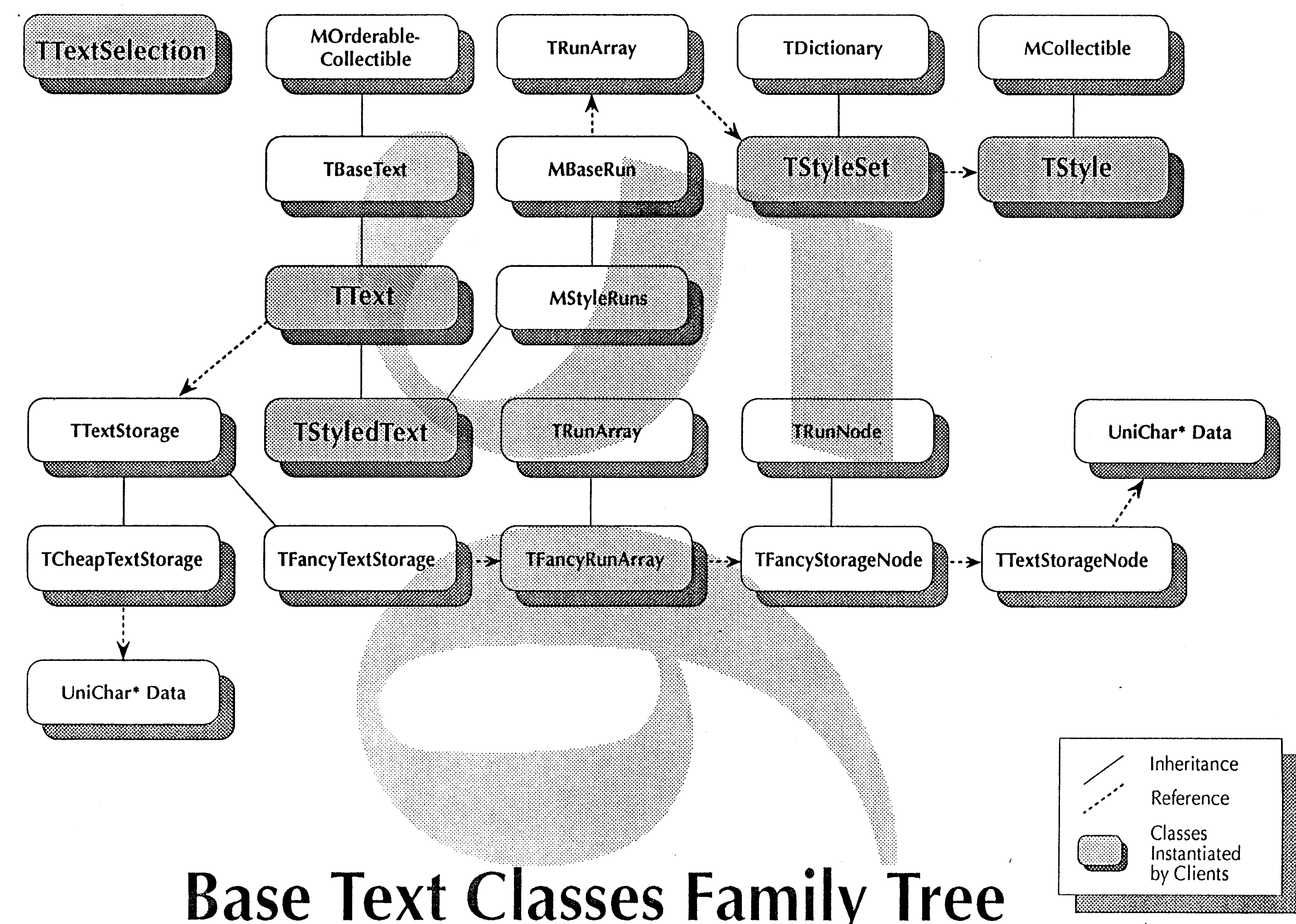
```
TTextSelection aRange(0, 3);
text->SetStyleInRange(ss, aRange);
```

Open Questions

- What types of operator overloading do users wish to see? We believe that assignment and indexing would be useful, and are included in this specification. Are there some others (concatenation, auto-increment, etc.)?

- What should I do about discontiguous selections?

- What should we do about comparisons? There are "degrees of matchingness" specified in the International Specification; how are we going to integrate this into the text stuff? Do we define behavior that includes some matching on the clients of a TBaseText object, or do we handle just text comparisons?



Base Text Classes Family Tree

ZZText

Text Formatting & Editing Classes

56

ZZText ERS

Text Formatting for Pink

This document describes the text editing facilities available under Pink. The system comprises complete text manipulation and formatting controls comparable to those found on the best composition systems. A client desiring text processing capabilities in an application should need do no more than instantiate a ZZText object, constrain it a bit, then "turn it loose." The classes discussed in this ERS are built upon the Base Text Classes, which have their own ERS.

Version for Big Pink Three

Roger Webster
x48115

56

Table of Contents

Introduction. 1

Philosophy. 1

The Model. 1

Theory of Operation. 2

List of Features. 4

Class Taxonomy. 6

Class Hierarchy Diagram. 8

Views, Galleys, and Text Relationships Diagram. 8

Sample Inheritance Diagram. 9

MBaseTextView. 10

TTextView. 10

TStretchableTextView. 11

TPageView. 11

TTextLine. 11

TParser. 11

TTextElement. 12

TProperties. 12

TSimpleParagraph. 13

TAdjustedText. 13

TGalleyText. 13

TEditableText. 14

TEditableSimpleText. 14

TEditableGalleyText. 14

TGalleyManager. 15

TEditTextSelection. 15

Examples. 15

Open Questions. 15

56

Introduction

This document is the ERS for ZZText, the text formatting system for Pink. In its final form, this ERS will represent the services available to clients wishing to support text manipulation. ZZText embodies sophisticated text handling features currently available on the most expensive and powerful publishing-oriented systems. In addition, ZZText provides a simple interface for basic on-screen text entry, facilitating dialogue box input, etc. This document will *not* contain a precise description of the methods in the various classes that ZZText comprises; that information is part of the header files for the text classes.

A successful implementation of ZZText includes enough functionality to satisfy the most demanding traditional publishing customer without penalizing simpler textual applications such as HyperCard. ZZText is an "Industrial Strength" text management system built upon a set of base text classes (see the *Base Text Classes* ERS), which support dynamic arrays of characters and styles. In addition, ZZText makes extensive use of the services provided by the line layout classes.

The distinction between ZZText and the line layout classes is important: ZZText is a *client* of the line layout services, using them to measure the width of text between two offsets, determine highlight regions, caret positions, as well as a plethora of international services including contextual letter forms, reordering (for languages such as Arabic), and mixed-direction text. ZZText provides mechanism by which lines may be generated and arranged on a page; it is not involved at the character interaction level, which is the domain of the line layout classes. Note that both ZZText and the layout classes are clients of the base text classes.

Philosophy

ZZText controls the appearance of text, specifically the layout of text (on a page, for example). One aspect of ZZText that permeates this entire document is a particular philosophy concerning text formatting: fast formatting results in poor quality, and good quality formatting is not fast. However, a good text processor supports both goals (fast and good). To get good text, the model must allow the specification of sufficient constraints on the text such that they satisfy the most demanding user. Less demanding users need not constrain the text as completely as more particular users.

The belief that *good* and *fast* are, for practical purposes, mutually exclusive, has led to the consideration of a two-pass approach to formatting. The first pass is the *fast* pass, and is defined as the pass that can keep up with a fast typist. The second pass is the *good* pass, and is defined as the pass that makes the Seybolds happy. The second pass consists of potentially more than one actual pass through the text.

The part of the second pass that is concerned with optimizing line breaks within a paragraph is based on Knuth's TeX method for breaking paragraphs into lines. In addition, this algorithm is restricted to languages that have "words," or some reasonable substitute. Some languages have no determinable word breaks, or they are very difficult to define (*e.g.,* Thai), and for these cases the algorithm cannot be used. Most modern languages lend themselves to this approach.

The Knuth method yields optimum line breaks for any given body of text, but can be extremely expensive computationally. Further, the algorithm is computationally unpredictable. Such time consuming tasks under Pink can be implemented as unobtrusive, low-priority tasks that tighten up the text while the user is concentrating on another part of the document.

One aspect of ZZText that will strongly reflect on the flexibility of text management and formatting for Pink is the ease with which new constraints can be added, as in class *TProperties*. Supporting an extensible set of properties is a major goal of this design, as it is impossible to anticipate everything clients are going to want to be able to do to text.

The Model

This section discusses both the model and some aspects of views of the model supported by the ZZText text building block. A thorough understanding of the model on which ZZText is based will aid potential clients of ZZText with respect to customization and utilization. ZZText is a set of classes that supports text editing, formatting, and page layout. The actual text component of a ZZText object is represented by one of the base text classes, normally *TStyledText*. Paragraphs are structures in a layer superimposed over the text that constitutes each paragraph.

Note that all internal measurements in ZZText are in printer's points and fractions thereof. There are 72.27 points per inch. Note also that a "units conversion class" is provided: *TMeasure*.

The model of the text managed by ZZText is an extension of the classes described in the *Base Text Classes* ERS. ZZText adds to this base-level functionality the concepts of paragraphs, lines, for-

matting, views of the text, text flows, and classes to support pagination.

ZZText does not implement any "user views" or presentations of the text. ZZText does not create windows, and does not directly support multiple views of a text object (yet). ZZText does not directly support even a single view of text, *from the user's perspective*. ZZText supports *TTextView* objects and through them provides an off-screen drawing feature. Text view objects provide the "mold" into which text is "poured," and conceptually they form part of the *model*, not a true *view* of the model. TTextView objects may be linked together to direct the flow of text, and may be contained within an object of class *TPageView*.

ZZText manages "flows" of text. Such flows are contiguous runs of text data with their attendant style and paragraph information. ZZText supports multiple independent flows, each of which may "flow" into multiple *views* (roughly analogous to columns). For the purpose of this document, such a collection of text is referred to as a *galley*. Galleys are peers, linked together allowing ZZText to manage more than one. This is *not* the same as a list of views linked together for the purpose of defining a text flow. Think of the root of this system as a list of galleys, each galley flowing into a linked list of views (the galley is the model, the views are simply that: views). ZZText keeps track of the *active* galley (the one where editing is taking place).

Although the term *paragraph* is used extensively in this document, more accurate terms might include *block, chunk, or group*, as paragraphs generally have the most meaning in documents containing horizontal text. Since ZZText enforces no directionality on text, and does not enforce any particular semantics for a paragraph other than a character grouping device, the ZZText "paragraph" is suitable for many uses. The base paragraph class *TParagraphs* a protocol-only class. As an example, the *paragraph* classes could be used in an MPW-style editor to implement lines.

ZZText paragraphs generally refer to a style sheet (a.k.a. a *property sheet*) to determine their characteristics, such as indentation, margins, default font family and style, etc. This separation of paragraphs and the properties associated with them makes the paragraph object a more general and much lighter weight object than a fully loaded paragraph would otherwise be. Also, it allows many paragraphs to share the same properties.

To support the editing of text with respect to insertions and deletions such as copy and paste commands, the class *TEditableText* defines behavior related to such editing changes. This class supports multiple discontiguous selections (although such functionality may not be implemented in the first version of ZZText), tracks the insertion point, and cooperates in the handling of commands (yet to be defined). Other types of editing changes (margins, etc.) are supported by other aspects of the ZZText classes.

ZZText provides a text manipulation environment that supports sufficient control over the appearance of text that almost any desired typographic effect can be achieved. The controls provided by ZZText allow hanging punctuation, drop capital letters opening a paragraph, etc. If there is something that ZZText does not support, a client has only to add new properties and override some of the formatting routines. (This is certainly a goal. It is stated here as if it were fact.)

In addition to the sophisticated text formatting capabilities discussed in this section, ZZText also provides a lighter-weight method for editing text that has minimal appearance parameters, such as the text in a dialogue box.

One type of functionality that ZZText will provide is the ability to identify "bad" lines. If, after the beautification passes, there remain lines that are too tight or too loose, ZZText will allow clients to locate such lines. Many high-end systems support this feature, and many demanding users require it. I often wish that FullWrite or other Mac text processors could do this, as it obviates the need to examine *every single line in the document*. After all, computers can do this sort of thing *so* much better...

Theory of Operation

A client will instantiate a *TEditableGalleyText* object using normal C++ programming methods. A *TEditableGalleyText* object will have certain default characteristics, including a paragraph style sheet specifying a font, style, etc., as well as certain default behavior. The default behavior of an editable text object will include keystroke handling, selection, cut, copy, paste, etc.

A client of ZZText also should establish a view (which must inherit from *TTextView*) in which to fit and display the text, though ZZText will manage text and keep it "unformatted" if no such view is provided. Also, at this stage in initialization, a client should establish a "page view" if one is desired.

Note that ZZText supports but does not require views of the text. If one or more views are defined for the text, those views will determine how the text is broken, otherwise the text will remain unformatted. Also note that this mechanism should

not be confused with a scheme to show more than one view of the same text (still being designed).

In addition, a mechanism will be provided allowing text to be broken into lines to determine its height, etc., without requiring that views or lines be instantiated.

For ZZText to handle multiple independent galleys and their attendant views, ZZText introduces the concept of a *Galley Manager.* The purpose of this object is to determine which galley in a multiple-galley text arrangement (such as a newspaper or magazine) is the "active" one, which galleys are current and that need updating, etc.

Typically, the classes that a client deals with depend on the level of sophistication of the application for which text is being used. The "dialogue box" case requires the creation of a single object of class *TTextView* and a single object of class *TEditableSimpleText.* For more complex applications, a client will likely use one or more objects of class *TTextView* (possibly linked together), optionally one or more objects of class *TPageView*, one object of class *TEditableGalleyText* per article or text flow, and one object of class *TGalleyManager* to manage multiple independent text flows (this one is optional if there is only one text flow, but recommended).

Application category	**Text classes used by clients**
Simple editable text, such as dialogue boxes and simple textual applications, possibly including text in cells of a spreadsheet, etc.	*TTextView (one)* *TEditableSimpleText (one)*
More complex textual applications, including text processors and any application requiring complex text adornment (multiple paragraph styles, etc.).	*TTextView (one or many)* *TPageView (one or many)* *TEditableGalleyText (one or many)* *TGalleyManager (one)*

List of Features

Feature	Probable Pass
General Pagination	
Force page break	*Both*
Prevent page break	*Both*
Force column break	*Both*
Group objects	
Separate objects	
River detection and avoidance?	
General Editing	
Selection	*First*
Multiple Selections?	*First*
Typing	*First*
Backspacing	*First*
Cutting	*First*
Copying	*First*
Pasting	*First*
Undo	*First*
Font and style changes	*First*
Widow and orphan control	*Second*
Number of allowable widow and orphan lines	*Second*
Depth (last line) matching & Alignment (all lines) matching ?	
Vertical justification	*Second*
Facing pages must match	*Second*
Columns must match	*Second*
Baseline grid	*Second*
Pages in a chapter (etc.) must match	*Second*
Lines must back up on pairs of overleaf pages	*Second*
Column balancing	*Second*
Keep with next paragraph	*Second*
Footnotes ?	
Placement	
Multiple line footnotes	
Carryover to following pages	
Line layout	*Both*
Multi-directional text	*Both*
Device-dependent character spacing	*Both?*
Marginal notes and illustrations	*Second*
Running headers and footers ?	*Second*
Variables, such as page number, date, etc.	*Both*
Multiple columns *	*Both*
Figures (probably a client-implemented feature)	*Both?*
Placement: exactly with text	
At top of page	
At bottom of page	
On a figures page	
Anywhere on a page	
Maintenance of sequence relative to mention in text	
Placement on facing pages	
Illustrations	*Both?*
See list above for figures	
Tables (Not at first)	*Both*

Equations, formulas (Not at first). . . . . *Both*
Cross-references (Not at first). . . . . *Second*
Bibliographical references (Not at first). . . . . *Second*
Tracking*. . . . . *Second*
Kerning (Line layout and ZZText) *. . . . . *Both**
Margins and gutters *. . . . . *Both*
  Left and right-hand pages. . . . . *Both*
  Gutters. . . . . *Both*
Paragraph Justification*
  Left*. . . . . *Both*
  Right*. . . . . *Both*
  Center*. . . . . *Both*
  Middle quadding*. . . . . *Both*
  Last line quadding options*. . . . . *Both*
  Justified*. . . . . *Both*
  Optical alignment* (Line layout). . . . . *Second*
  Hanging indentation*. . . . . *Both*
  Knuthian last-line handling*. . . . . *Second*
  Non-Knuthian last line handling*. . . . . *Both*
  Suppress line break*. . . . . *Both*
  Discretionary hyphenation*. . . . . *Second*
  Ladder suppression (Knuth)*. . . . . *Second*
  Esthetic ragged-right margins. . . . . *Second*
Layout grid support ? (Jane?)
Style Sheets *. . . . . *Both*
  Hierarchical?. . . . . *Both*
Tabs *
  Left*. . . . . *Both*
  Right*. . . . . *Both*
  Center*. . . . . *Both*
  Decimal*. . . . . *Both*
  Arbitrary character*. . . . . *Both*
  Various leader options*. . . . . *Both*

Class Taxonomy

Class Name	Description
TStyledText	From the *Base Text Classes*, this is the class that manages the actual text and styles associated with a ZZText object.
TAdjustedText	Implements automatic index adjustment to account for movement through text (due to insertions, deletions, arrow keys, etc.).
TProperties	Repository for information controlling the appearance of text in a paragraph, such as indentation, flush parameters, etc.
TTabs	Specification class for tab stops. This class support all of the standard types of tabs, and the *TProperties* class contains a pointer to an object of this class. The full definition of this class requires more discussion with the International Group.
TChangeInfo	Class that holds change information supporting incremental updates and interruptable drawing.
TParagraph	Protocol-only class defining the behavior of both simple and fancy paragraphs.
TSimpleParagraph	Subclass of *TParagraph*, with a reference to a *TProperties* object. The pointer to the properties object may be *nil*, in which case certain system defaults are used. This class is primarily for use in a situation where "cheap" editable text is needed, as in a dialogue.
TFancyParagraph	Subclass of *TSimpleParagraph*. This class adds behavior specific to high quality paragraph formatting.
MParagraphs	Class implementing behavior based on *MBaseRun* (see Base Text Classes) specific to runs of paragraph information. Fairly low-level implementation-specific class.
TGalleyText	Layer of information over a run of text that specifies the beginning of each paragraph. This class is a composite of *TStyledText* (see Base Text Classes) and *MParagraphs*. This data structure also may be thought of as a *text flow*, and another name for this class might be *TStyledParagraphs*.
TEditableText	Specifies behavior related to insertions and selections. Allows place holding in objects derived from class *TBaseText*.
TEditableSimpleText	This class implements editing text that has no paragraphs (or all text is in one paragraph), such as a dialogue box. It does not incur the overhead required by text that has multiple paragraphs and property sheets. *N. B.* Multiple paragraphs may be simulated with this class, as it does support "hard returns" in the text. Such returns force a new line, and result in indentation of the new line. No run array is required for paragraph maintenance, but only one property sheet is allowed for all of the text in a *TEditableSimpleText* object.
TEditableGalleyText	Companion class to *TGalleyText* and *TEditableText*. This represents the "standard" ZZText text editing object.
TEditableFancyGalleyText	Class identical to the one above, but automatically starts a background task to accomplish fancy formatting, including paragraph line break optimization.
TGalleyManager	Class that handles multiple, independent instances of *TEditableGalleyText* objects. Supports multiple flows of text, such as articles in a newsletter.
TTextLine	Component of a formatted paragraph. A light-weight object used to render the text of a paragraph.
TTextElement	This class is used to stored information about a paragraph that has been "decomposed" into the atoms it comprises. This structure is used to cache information such as the widths of elements within a paragraph. Such objects are also the basis of Knuth-style text formatting and greatly speed the processing of text

with Knuth's algorithm as a fair amount of time spent formatting text in TeX is allocated to parsing the input. Both *fast* and ***high quality*** text formatting algorithms can make use of this concept. Also, by isolating the intelligence required to decompose paragraphs into words in an object such as a parser (see below), internationalization is simpler.

TParser .... Class whose function it is to tokenize the text associated with a paragraph. For English, this amounts to breaking the text into words (and punctuation, if hanging punctuation is desired).

MTrackable .... Mix-in for views supporting tracking protocol. This class provides common protocol for all views that may become trackers (for "handing-off" of tracking in multiple view situations, etc.). This class should be mixed in to any view that may participate in tracking the mouse.

TTextTracker .... Class used to track the mouse around multiple text views. An object of this class keeps a pointer to the view that created it, and calls methods in that view as tracking proceeds. Whenever this tracker's view gives up its tracker status and another view takes over, this tracker deletes itself.

MBaseTextView .... Class implementing common functionality shared by *TTextView* and *TPageView*, chiefly in the form of linking these objects to one another (previous page, next page, etc.). Also supports shrinkability and stretchability protocols.

TTextView .... Class implementing the area into which text should be set. Roughly analogous to a column in many situations. The interaction between this class and the *TParagraph* class determines line breaks within a paragraph. If a *TPageView* object is instantiated for this view, then the view coordinates are relative to the page that encloses it.

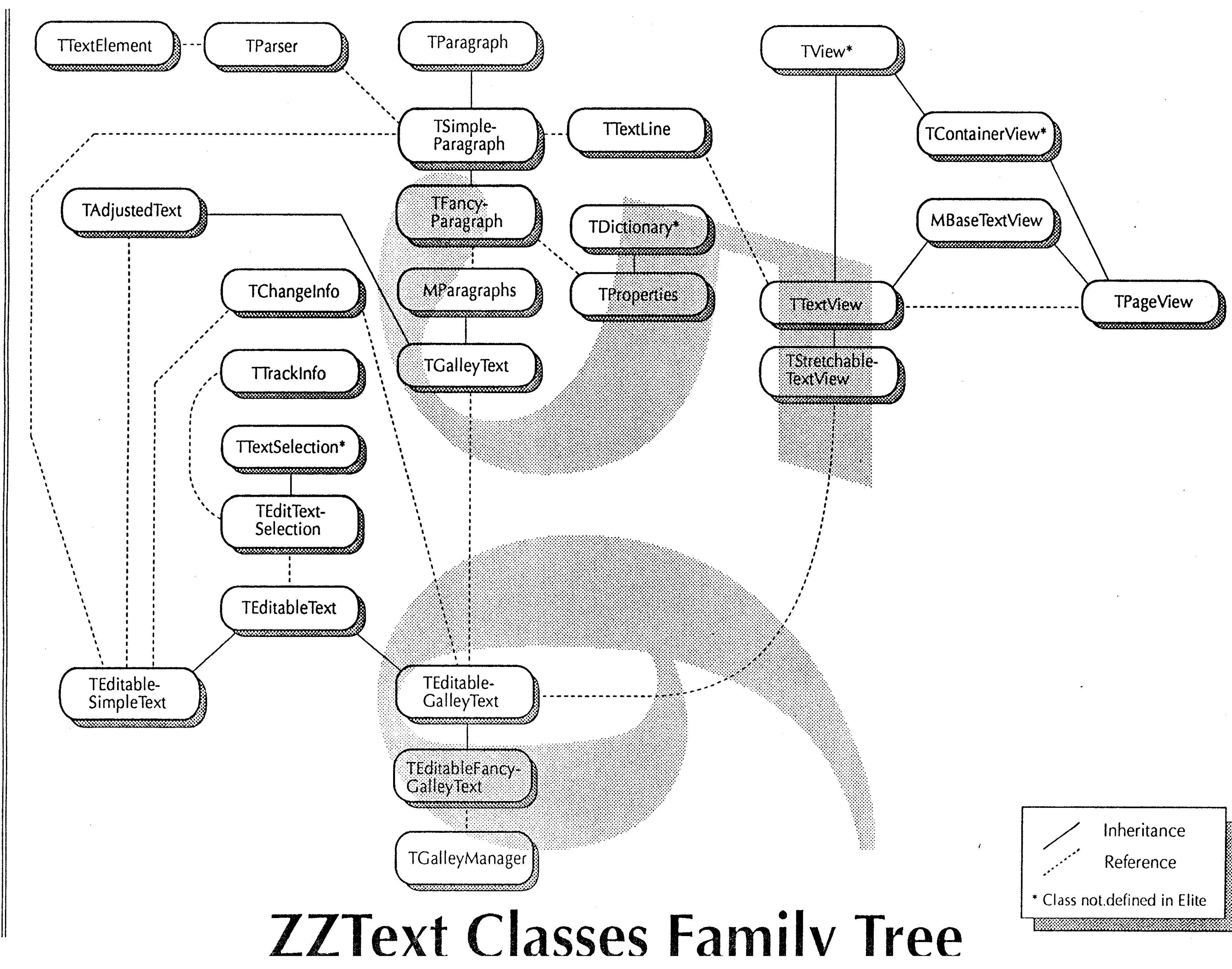
TStretchableTextView .... Very similar to *TTextView*, which is its superclass. The principal difference is in the handling of overflow conditions. This class implements behavior that creates stretches itself.

TPageView .... Class that groups TTextView objects (e.g., for columns) and supports odd/even pages, facing and backing pages, etc. It is important to note that although ZZText supports the concept of a page, it *does not require that a page object be instantiated*. The page reference in a *TTextView* object may be nil.

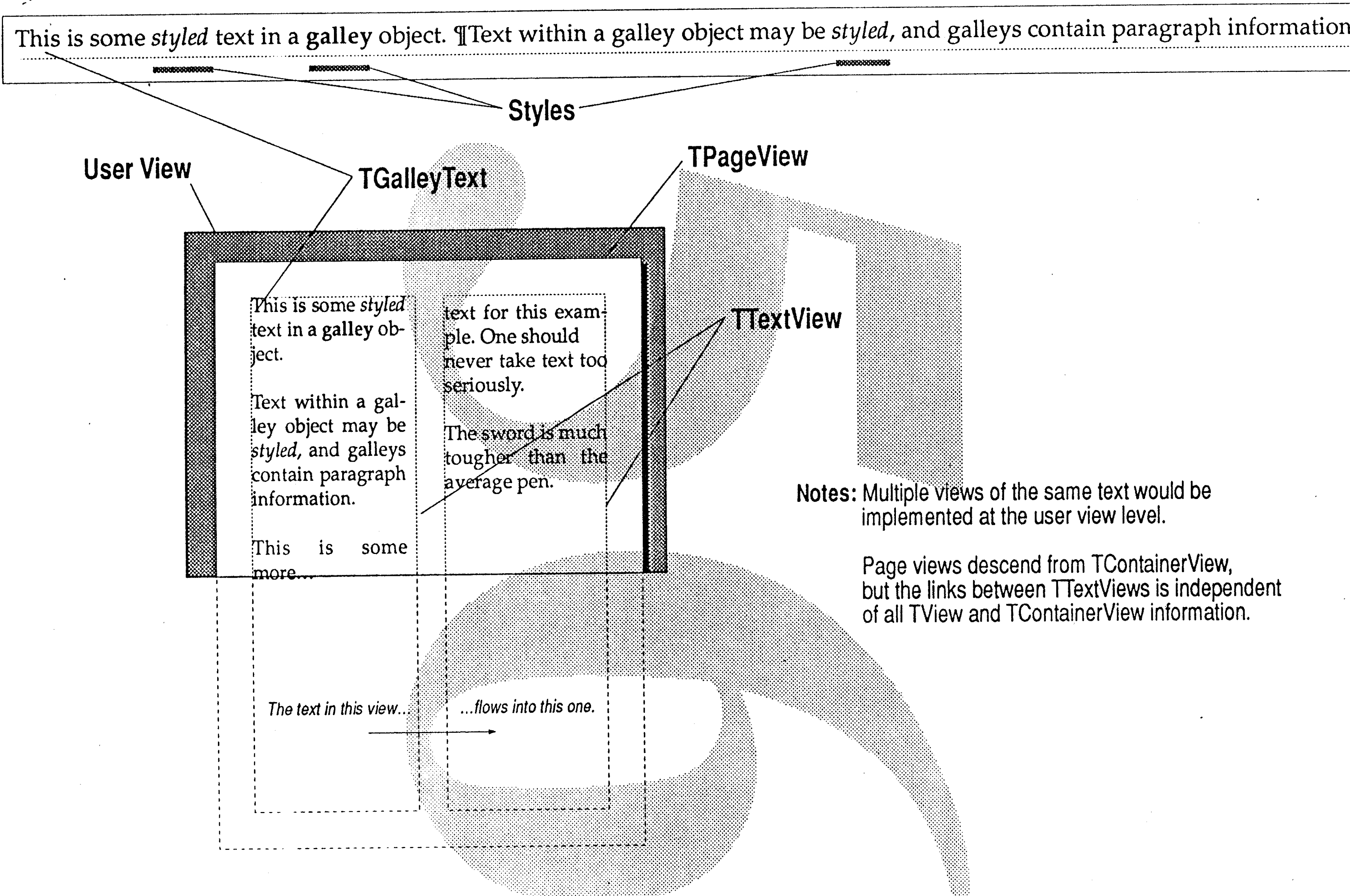
TEditTextSelection .... This class supports some of the command creation and text editing functionality required by a text building block. It is a descendant of class *TTextSelection*, and supports basic operations on text, such as *cut, copy, paste,* and *insert*. The exact specification of highlighting will not be completed until some of the user interface issues are resolved, such as what to do about direct manipulation of text. For the sake of this ERS, I assume that there will be, at a minimum, some type of "on," "off," and "dim" highlighting protocol, and that it may be up to ZZText to define the protocol for blinking the insertion point.

TTextCommand .... Base command object for text editing. Undefined for now, but will likely descend from *TCommand* or something similar. The exact definition of this class will wait, pending work by Arn and Larry in the areas of *undo, redo,* and *voodoo.*

TCutTextCommand,
TCopyTextCommand,
TPasteTextCommand,
TStyleCommand,
TFormatCommand,
TInsertTextCommand .... Refined command objects base on *TTextCommand* (above), implementing the various text commands. These classes will support the Pink "clipboard."



ZZText Classes Family Tree



Notes: Multiple views of the same text would be implemented at the user view level.

Page views descend from TContainerView, but the links between TTextViews is independent of all TView and TContainerView information.

Views, Galleys, and Text

MBaseTextView

This class implements behavior allowing a view to be linked to views logically "before" it and "after" it. Both *TPageView* and *TTextView* descend from this class. Conceptually, a descendant of this class may represent either a column of text or a page consisting of columns of text (but not both simultaneously). Linked objects of class *TTextView* are analogous to columns, while objects of class *TPageView* are pages, or groups of columns.

One area requiring great flexibility when handling text is what to do when text hits the edge of the view, whether the edge is a side or the bottom. In order to provide the greatest functionality and flexibility, ZZText supports two concepts: the *stretch limit* and the *grow limit*. When a view overruns the stretch limit, the method *StretchLimitOverrun* is invoked. When text exceeds the grow limit, the method *GrowLimitOverrun* is called. In many cases, the stretch limit is set to the grow limit, and one of the methods does nothing. There are also analogous methods for underflowing a view.

As an example, consider MacDraw: when a user clicks to place a text object, a box is drawn around the insertion point. As the user types, the box expands both horizontally and vertically (*e.g.*, when a return is typed). When the user "runs into" the right edge of the paper, a new line is automatically begun. When the user runs into the bottom of a page, a new page is allocated and typing continues on the new page. For the sake of the analogy, the automatic stretching of the box that outlines the text is the "stretch limit," and the automatic creation of a new page occurs when the text exceeds the "grow" limit.

Note that this approach makes the most sense if the region into which text is being fitted is rectangular. Also note that this has nothing to do with text that gets pushed into a new region as a result of an insert *if that region already exists*. If an "overflow" region is available (the is a view in the chain following the current viewl), then the current view is not grown and text is "flowed" into the available region.

For text processors, the default behavior will likely be not to grow or stretch a view when it is overrun, but to put the overflow text into the next view in the chain of views. This behavior is automatic if the pointers to the stretch limit and grow limit rectangles are both nil. There is a subclass of *TTextView* called *TStretchableTextView* that implements the stretching and growing behavior.

TTextView

This class supports the concept of an area in which text is set (*e.g.*, a column). Text may *flow* into a text view from another text view, or the flow may originate in this view and flow into following views. In addition, several provisions exist regarding the disposition of text that does not fit into a text view, including overflowing to another view, enlarging the current view (see *MBaseTextView* above), clipping, etc.

In essence, a text view is analogous to a *column*, though a text view is more flexible than a column (the text view definition includes an optional region). A text view may be of any shape and have any directionality, though only rectangular views are implemented at present.

A text view is *not* a paragraph. Properties of the text itself are defined in other classes, such as *TParagraph* and *TProperties*. A text view controls more where text may go, and has a less direct effect

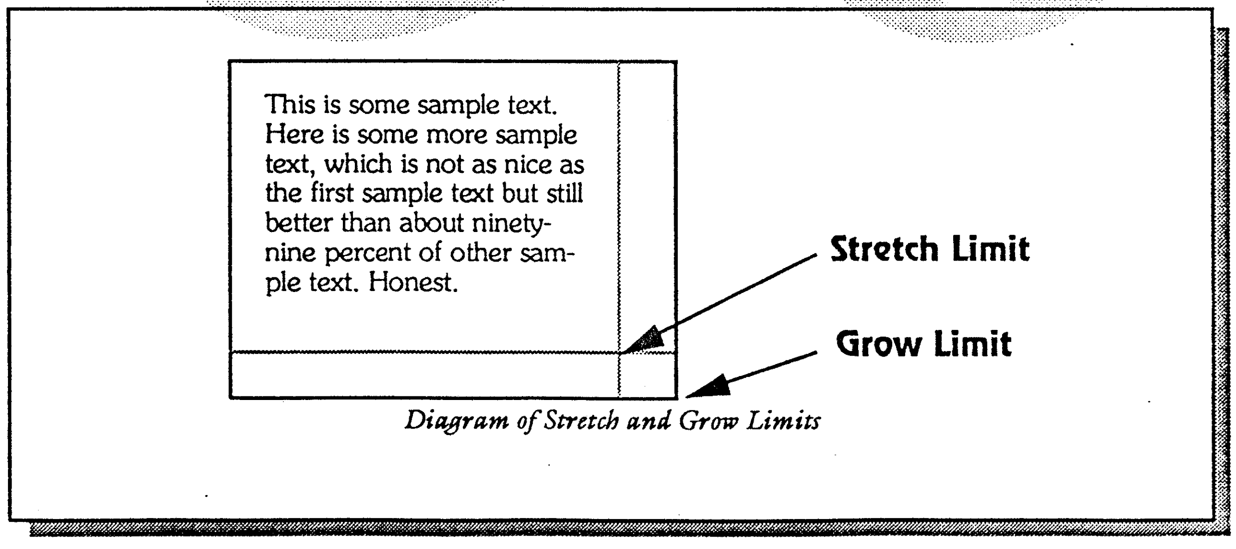


Diagram of Stretch and Grow Limits

on how it will look. In addition to the properties of a text view, other views may define a relationship to this view. This allows such functionality as automatic adjustment of the origins and sizes of views when footnotes are added, as well as aligning baselines in adjacent views.

Lines belong to paragraphs (which generate them), but are used to present the text given a particular view. Normally, a view neither creates nor deletes lines (this is done by paragraph objects exclusively). However, it is desirable to maintain lines without necessarily maintaining paragraphs. For this purpose, views may occasionally manage line objects directly (they never create them, though they may delete them some time after the paragraph that generated a particular line has been deleted).

Text views directly influence how text is broken by defining the boundaries into which the text must fit. In addition, text views control how existing lines shift up and down as text is added or removed. In particular, pushing lines into following views and pulling them back is handled, in part, by the view itself.

If there is a shape region, the text is fitted *into* the shape defined by the region (this is not yet supported). This region is the result of region operations that define the shape the text should take, not the shape or shapes the text should *avoid*. An example of establishing three views, all linked together, follows:

```
TGPoint sizeAB(200, 145);
TGPoint sizeC (415, 200);
TGPoint locationA(5, 0)
TGPoint locationB(220, 0)
TGPoint locationC(5, 155);
TTextView* aView = new TTextView(sizeAB,
    locationA);
TTextView* bView = new TTextView(sizeAB,
    locationB);
TTextView* cView = new TTextView(sizeC,
    locationC);
aView->SetNext(bView);
bView->SetPrevious(aView);
bView->SetNext(cView);
cView->SetPrevious(bView);
```

TStretchableTextView

This class is essentially identical to its superclass (*TTextView*) except that this class implements the strech and grow protocol described above. Note that stretching and growing is not the default behaviour of *TTextView*, though the paragraph classes always attempt to stretch a view before overrunning it, if there is no next view in the chain.

TPageView

This class is a container view for objects principally of class *TTextView*. Its chief purpose is to group objects of class *TTextView* together in a way that allows computation of such typographic refinements as matching column depth, overleaf page depth matching, etc. Note that this class is a subclass of *TContainerView*, rather than *TView*. This is a result of the notion that pages are simply view objects that clip elements to their boundaries. In an application such as MacDraw, this is not necessarily the case.

TTextLine

An object of class *TTextLine* is the principal unit for rendering data in ZZText. Objects of this class have pointers to the lines that precede and follow them in their paragraph, as well as the lines that precede and follow them in the view. Note that since a paragraph may reside in more than one view, it is possible for these two pointers not to match (if the line is the first in the view but not the first in the paragraph, then its previous line pointer will be non-nil, while the pointer to the previous line in the view will be nil.

TTextLine objects are also used to keep track of change bars, if that functionality is enabled. Change bar data is managed automatically, as the mechanism for rebreaking only flags lines as different if they needed to be rebroken. In addition, information about the desired "quadding" (flush parameters, centering, etc.) of the line is passed to it upon creation by its paragraph.

Line objects cooperate with the line layout classes to enable rendering and highlighting.

TParser

This class is responsible for decomposing the text of a paragraph into "words." In early versions of ZZText, this will support only languages with relatively easy-to-recognize word delimiters (like English and spaces). It also recognizes punctuation. The interesting thing about this class is that it is incremental. That is, given an arbitrary insertion or deletion (or combination), this class performs the minimum work necessary to restore the target paragraph to a state of full tokenization.

This method is extremely fast, and has two positive results: the tokenized paragraph can be reformatted much faster in the simple case (adding up widths until the line overflows) as only a few additions are needed per line. In addition, since Knuth's paragraph breaking algorithm uses tokens, the paragraph is already in a handy state for "optimization."

The incremental aspect of this class is carried one step further: it can split a paragraph into two paragraphs by retokenizing the affected text and then moving tokens following the paragraph break into a list of objects for the new paragraph. In addition, a parser also may join together the token lists of two paragraphs that become one as a result of a deletion. This is analogous to the split case: the text is retokenized and the tokens from the second paragraph are added to the list for the first. The second paragraph is then deleted normally.

TTextElement

This class implements text tokens for formatting. Basically, tokens are words (in languages that have the concept of a word; notable exceptions include the Thai script). Each word roughly corresponds to a group of character boxes in Knuth's box-penalty-glue model, strung together to form one large character box (*i.e.*, a *word* box). Also, punctuation and other separators (such as spaces) are considered text elements.

Tokens maintain an index relative to the paragraph to which they belong. Also, a token contains its width, height, baseline, the width of any trailing white space, and the length (in characters) of the token. Note that tokens do *not* contain the index of the first element in the token. The index of the current and previous tokens are always maintained by a *parser* object, making this index information redundant. By omitting this information the indices need not be adjusted when inserting or deleting text in a paragraph, thereby making the retokenization process very fast. Tokens are created and destroyed by an object of class *TParser*.

The entire token list for a paragraph may be discarded, if necessary, for space considerations. A paragraph that does not have a token list is tokenized automatically when necessary. Please see the description of the *TParser* class.

TChangeInfo

This class implements a cache of information needed by the pagination methods. It does all records-keeping required by the incremental reformatting code. Among the elements kept track of by this class are the index of where reformatting should resume following suspension to handle user input, the range of text affected, whether or not the reformatting process is interruptable, etc.

This class also remembers where drawing was suspended when lines are being drawn into a *TTextView*. Again, this occurs as a result of user input interrupting the reformatting code. Note that *both* the reformatting process *and* the redrawing process are incremental and interruptable. If the change affects the document in a way that is not interruptable, it is normally the responsibility of the client's code to disable formatting/drawing until the critical section has been updated. Note that the interface for controlling interruptability, etc. is part of other classes, such as *TEditableText*.

TProperties

The properties of a paragraph that control the way a paragraph looks are not stored with the paragraph. Instead, they are stored in a paragraph property sheet, one of which may suffice for many paragraphs.

Rather than attempt to implement all of the properties that any client could ever envision (an impossible task), there is instead an extensible property mechanism provided that allows the set of paragraph properties to be expanded as needed. The class *TProperties* is a subclass of *TDictionary* (see the Utility Classes) that matches names (tokens) with values. The token definition is in *Layout.h*, the header for the layout classes. Property values are accessed via syntax such as:

```
flushLeftStyle = properties->
     PropertyValue(kFlushLeftToken);
flushLeft = flushLeftStyle->Value();
```

Generally, properties must be "known" to the formatting engine for them to be applied. Should a client wish to add a new property previously unrecognized by the formatting engine, the client must override some method to apply the new property.

Note that some properties, such as line spacing, may be composed of several "properties" in a *TProperties* object. Line spacing has four values governing its behavior: a leading constant, a "natural" line spacing multiplier, and minimum and maximum line spacing values (in points). These values are used according to the following formula:

*«Minimum line spacing» < («line spacing multiplier» * «natural line spacing») + «leading constant» < «maximum line spacing»*

Note also that the arithmetic for calculating an unconstrained line spacing value is a first-order polynomial (of the form $y = mx + b$), where x is the natural line spacing derived from the height of the tallest font on the line (*e.g.*, ten points), b is the leading value (*e.g.*, two points), and m is the line spacing multiplier (*e.g.*, 1.2). For this example, y would be 14 *points* ($y = 1.2 \times 10 + 2$).

TParagraph

A protocol-only class defining the basic behavior of paragraphs in ZZText (note the protected constructor in the class declaration).

TSimpleParagraph

This class supports a relatively light-weight paragraph with respect to the number of member fields. "Simple" is somewhat euphemistic, as this class implements all protocol for breaking a paragraph into lines, albeit simplistically (hence the name). The basic algorithm for line breaking is first to make sure that the token list is up-to-date by incrementally retokenizing the paragraph to accommodate the most recent editing changes. The line before the first line enveloping the change is found and rebroken (so that words that get pulled onto the previous line are handled properly).

At this stage, a guess is made to determine in which *TTextView* lines should go (this guess can be wrong, but the error must be to put the line into a view that comes *before* the correct view if a wrong guess is made. This wrong guess is detected when it is determined that there is insufficient room remainging in the "guess" view.

System default properties are used when the paragraph does not have a property sheet of its own.

This class allows numerous small text edit fields in a dialogue box without incurring the formatting overhead needed to do high quality text formatting. Objects of class *TSimpleParagraph* are able to apply a "first-fit" algorithm to themselves to determine line breaks. The range of characters affected by the most recent change is contained in an object of class *TChangeInfo*, and while paragraphs are formatting themselves within this range, the formatting process is not interruptable. This is so that typing (or cutting and pasting) context can be preserved.

Choosing a long paste may result in some delay. The interaction between this class and *TTextView* represents the most complex area of ZZ-Text, chiefly by virtue of the fact that these two classes account for most of the code in ZZText (scrolling, painting, measuring, etc.).

In general, the formatting process generates lines as atomic units. As lines are generated and as the last character position *directly* affected by an editing change is passed, new lines are compared against old lines to determine when new lines are no longer being generated. When this occurs (lines have "re-synchronized"), formatting stops. If a client should choose to override this behavior, extreme caution is advised. Even if you know what you're doing it is easy to get this wrong.

TFancyParagraph

This object has nothing to do with calculating minimal updates for painting text as it is typed; that behavior is implemented in class *TSimpleParagraph*, the superclass of this class. The chief function of this class is to provide a "beautification" method that may be invoked on demand, or automatically via some "beautification process."

The superclass provides a fast, first-fit algorithm (stuff text until there's too much, then stretch the spaces to justify, with no hyphenation) to give the user an idea how the paragraph will look. Some time later, the beautification method is invoked when it will not interfere with the user, and a new "optimum fit" set of line breaks for the paragraph is calculated. The new line information is substituted for the old, and if the paragraph is visible, prepares a new image of the paragraph off-screen, and replaces the old paragraph image with the new image.

Remember that formatting text has no meaning if there is no view with which the text is associated (or a view surrogate). This results from the fact that the paragraph contains no information about the shape into which it must fit (since this may vary depending on the depth of the text). To understand why this particular structure was chosen, bear in mind that views can span paragraphs *and* paragraphs can span views.

TAdjustedText

This class automatically maintains an index adjustment for accessing style information that compensates for the direction in which the user was "moving" through the text For example, if the user establishes a style and begins typing, the direction is "forward." If the user types five characters that are inserted before existing text (where the old style begins), the current unadjusted index is five (the index where the next character will be inserted). The style at this index is the old one, not the new style established for typing. This class compensates for this condition, and others, including deletions and using arrow keys.

Clients should not normally deal with this class directly, or even be aware of its existence. This description is included for completeness.

TGalleyText

Clients will note that this class is derived from two classes representing styled text and runs of paragraphs. The implementation of this class is roughly similar to the implementation of style runs (please refer to the diagram).

It probably will be true that the algorithm for removing tokenized paragraphs from the tokenized paragraphs list will never "un-tokenize" a paragraph that is also on the ugly paragraphs list.

This is the class used by *TEditableGalleyText* to implement storage and run maintenance. Specifically, this class implements runs representing paragraphs (*complex* paragraphs—not the "fake" ones used in class *TEditableSimpleText*). This class controls the number of paragraphs that may be tokenized at one time, as well as maintaining the list of paragraphs that have not been "beautified."

TEditableText

This is essentially a protocol and support class forming the backbone of the other two editable text classes: *TEditableSimpleText* and *TEditableGalleyText*. It insures that the typing caret is displayed, and implements some of the common protocol shared by the more refined classes for insertions and deletions.

TEditableText maintains the *typing style*. This style controls the appearance of text as it is typed, if appropriate. In addition, corrections are made automatically to sense conditions such as deleting to the beginning of a style run. In this case, the client normally does not wish for the style run (which may now be of length zero) to disappear.

This class also contains an object of class *TChangeInfo*, which maintains the incremental update information used by the formatting code.

One other area of support provided by this class is in the area of idle time processing. Formatting and painting are interruptable and are resumed automatically whenever user input is not pending. If user input is not pending, the method *HandleDoIdleTask* is called, which in turn calls the method *TryToFinishFormatting*. This method must be overridden in subclasses (such as *TEditableGalleyText*) to determine if any formatting or updating remains.

TEditableSimpleText

This class implements a relatively lightweight editable text object. This type of text object should be used in the text edit fields of a dialogue box, for example. Paragraphs are not supported directly by this class, as they are implemented in classes that derive from *TGalleyText*, and this class has as its foundation an object of class *TAdjustedText*. Note that this class give the appearance of simple paragraphs by indenting the first line following a carriage return. At most one line in each pseudo-paragraph may be indented (note that *TEditableGalleyText* supports multiple line indentation, including hanging indentation).

All "paragraphs" in the text backing up an object of class *TEditableSimpleText* must share the same *TProperties* object. In reality, there is only one paragraph (just the appearance of multiple paragraphs) in this class.

TEditableSimpleText should be used in cases where the text is relatively short (dialogues, etc.) and minimal formatting is required. This class is not recommended for button labels, etc. (it's too expensive).

TEditableGalleyText

This class and *TEditableFancyGalleyText* are the principal classes used to manipulate single flows of text through one or more views (derived from class *TTextView*). This class insures that all paragraphs affected by an editing change are updated before control is returned to the client. This is not preemptive in that formatting and updating stop as soon as the client's most recent changes have been incorporated, at least locally, into the model.

Formatting can only be suspended at the end of a line, and only when all text *directly* affected by the editing change has been updated. For example, if ten characters are typed and the resulting change affects two middle lines in a twenty-line paragraph, formatting will not be interruptable until both changed lines have been rebroken *and* redrawn. Note, however, that even if all remaining lines in the paragraph (or even the entire text flow) are now invalid, if the user is still typing, changes to the rest of the document will only occur when they will not preempt the user.

Both formatting and drawing are atomic at the line level. After each line is broken (beyond the range of an editing change), the formatter checks state to determine if there is input pending. If there is, formatting is suspended and the new input is handled. Lines changed as a result of change propagation (change *rippling*) will not necessarily be draw immediately, but are flagged as "needing to be drawn," and another pass through the views during idle time draws flagged lines.

If a client desires only one text flow (through one or many views), this class is recommended. To use this class, establish an initial view (or views, please refer to the code fragment in the *TTextView* section) and do the following...

```
TEditableFancyGalleyText text(firstView);
```

...and that's it. The client has the option to specify a different default typing style (the system default style will be used in the absence of a client-specified style), and the default property sheet may be changed arbitrarily. For example...

```
sizeStyle = new TPoints10();
text.TypingStyle()->AddStyleToSet(sizeStyle);

THangCountStyle hang(1);
```

```
THangAmountStyle amount(24.0);
text.DefaultProperties()->AddProperty (hang);
text.DefaultProperties()->AddProperty (amount);
```

...sets the default type size to ten points and sets indentation at a third of an inch for the first line in each paragraph.

TGalleyManager

This class manages multiple text flows, such as the articles in a newspaper. For the sake of discussion, the "active galley" is the one with the blinking insertion point; the one into which non-positional events will be directed. The principle role of this class is simply to maintain a list of objects of class *TEditableGalleyText*, and to remember which galley is the active one.

This class gets involved in installing the correct selection object as the non-positional event target such that the "active galley" has the "active selection" (see below).

TEditTextSelection

This class is a subclass of *TTextSelection* (defined in the base text classes), and supports some of the editing operations typically done to text, including *cut*, *copy*, *paste*, *insertion*, and style changes. Class *TEditableText* maintains a list of selections (multiple, discontiguous selections are supported, although exactly how they will be used is still unclear).

All objects of class have at least one object of class *TEditTextSelection* in a list of selections. This selection always heads the list of selections, and can never be deleted or put anywhere in the list other than at its head. *The first selection in the selection list is automatically installed as the target for all non-positional events.* The reason that this selection object may not be deleted is that the event mechanism would not know if its target has been removed.

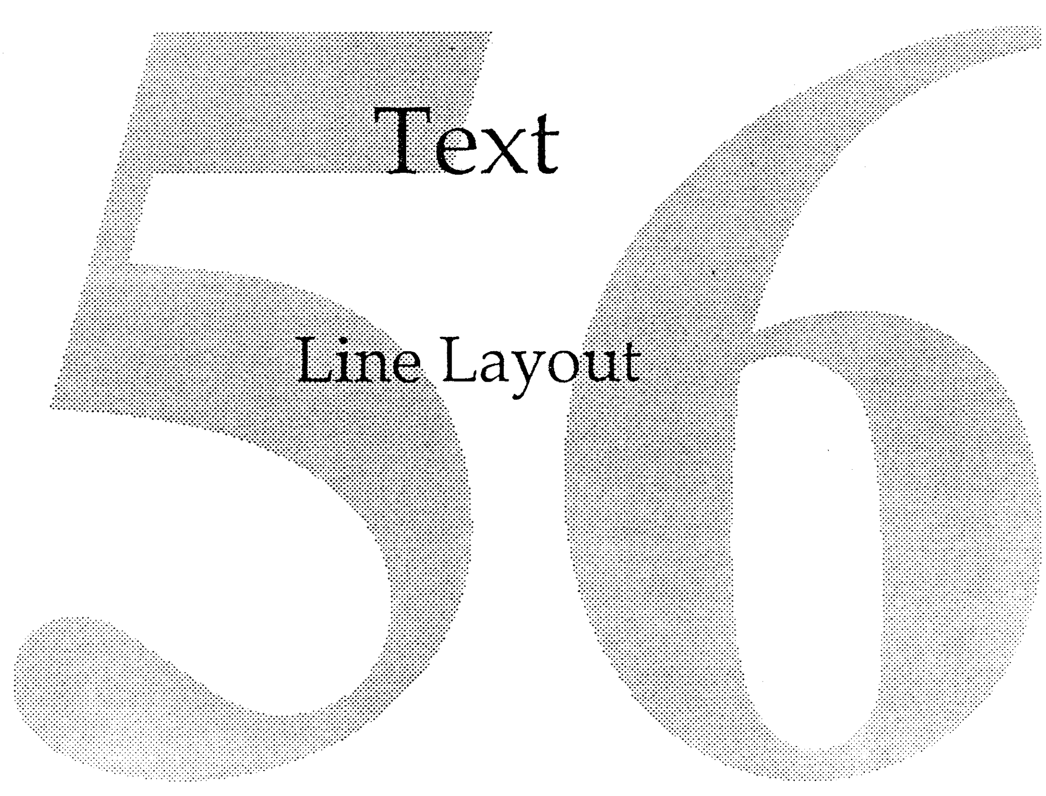
Examples

This section will be rewritten eventually.

Open Questions

- Tracking, selection and auto-scrolling?
- What about a system-level style manager?
- One item not covered in this document is how background tasks that have completed updating a part of the document (e.g., a paragraph) will synchronize the update of the display with the current user activity. Later.
- Headers, footers, footnotes, table of contents, etc. The specification of these will come following some more discussion with the Jane Folks. For some things, (table of contents), some type of active value mechanism may be a good idea.
- Commands and selections... what do they look like in the Pink programming model?
- Reading and writing text... will be covered soon.
- Tabs... are hard. The line layout manager provides much of this functionality.
- Copyfitting. One approach used by some magazines is to copyfit the story from the *end*, then see how much of it is left over at the beginning. This allows the title and any artwork/graphics to be height-tweeked to take up the slack. ZZText does allow access to the "room left over at the bottom of a text view." This value with respect to the last view in a list of views for a particular galley (article) could be displayed, and the user could adjust things until the room left over was zero.

56



Text

Line Layout

56

1. Overview 1
What Line Layout is 1
What Line Layout isn't 1
Why use Line Layout 1
2. Where Line Layout Fits into the System 2
Graphic System (Albert) 2
Low-level Toolbox 2
Fonts 2
High-Level Toolbox 2
3. Important Concepts 3
Characters, Glyphs and Unicodes 3
Fonts 3
Styled Text 4
Reordering of Characters For Display 4
4. Features of Line Layout 6
Graphical Features 6
Layout Features that Affect Glyph Mappings 6
Layout Features that Affect Glyph Positions 7
Glyph Adornments 9
5. Objects for Displaying and Hit-testing Text 10
TStyledText 10
TCaretPlace 10
TInsertionPoint 10
TLayout 10
6. How To Use Line Layout 11
Setting Up a TLayout 11
Displaying text 11
Highlighting 12

Hit-testing and Carets 13

Measurement 14

Justification 15

Line Breaking 15

7. System Defined Styles 16

Graphical Styles: 16

Styles that Affect Glyph Determination: 16

Styles that Affect Glyph Positions: 17

Glyph Adornment Styles 18

8. Extending the system 19

Additional Objects 19

TFont 19

TLayoutGraphic 19

What TLayout does 20

Adding New Features 21

Adding New Styles 21

Adding Embedded Graphics 22

9. Performance Issues 23

Caching of Information By a Client 23

Size 23

TLayout Speed 23

Fixed Point instead of GCoordinates 23

Appendix A— Concepts 24

Ligatures, accented forms and accent ligatures 24

Applied marks 24

Contextual forms 25

Automatic Kerning 26

Automatic Cross-Stream Kerning 26

Optical Alignment 26

Hanging Punctuation 28
Baselines 28
Character Reordering 29
Split Carets 30
Glyph Metrics 32
Appendix B — Application Design Requirements 34
Paragraphs, Tabs, and Rulers. 34
Multi-Columns Pages. 34
Documents and Page Numbers. 35
Mixed Scripts Within a Paragraph. 35
Appendix C — Unsatisfied External Requirements 36
Font Architecture 36
Missing Fonts and Missing Glyphs 36
Getting Metrics 36
Human Interface 36
Passing Application specific text styles 36
Validation of Line Breaks by an Editor 37
Appendix D — Open Architectural Issues 38
Styles 38
Simplified Programmatic Style Extensions to Line Layout 38
Line Breaking 38
Device Dependency 39

56

1. Overview

What Line Layout is

Line Layout is an extensible set of objects used to display single lines of fully styled text. Besides displaying text, Line Layout also provides methods that aid in the measurement, selection, positioning, and highlighting of characters. Line Layout is for the rare programmer who wants to write a text formatter. The avarage Pink application programmer should not use Line Layout, but should use 'higher level' objects such as ZZText objects for text display and editing.

A rich set of typographical formatting controls are handled automatically by Line Layout. The typographical control supported by Line Layout is a superset of the controls provided by high end programs available on the Macintosh.

Line Layout also provides script specific contextual formatting features offered by few other systems. This enables Line Layout to format text for all major writing systems. This includes formatting features found in Asian, South-East Asian, Middle Eastern and European scripts.

Line Layout handles text in one of two fundamental directions: horizontal or vertical. Within the horizontal direction lines can be formatted for right-to-left or left-to-right paragraphs—a distinction needed to handle Arabic scripts correctly when mixed with non-Arabic scripts.

What Line Layout isn't.

Line Layout is not a text editor or a word processor, although it should be used to implement one. It also has no "user interface" per se, since it's not an application.

Line Layout does notunderstand anything about the organization of text at a level higher than a single line or line segment. In particular, paragraph properties such as tab stops, "rivers of white space" or general paragraph appearance are not in Line Layout's bailiwick. Additionally, Line Layout does not decide line breaks, line positions, or line orientation (horizontal or vertical.). However, Line Layout does provide facilities that help text editors or higher level toolbox services deal with these functions.

Why use Line Layout?

The reason an application should use Line Layout is that it hides the complex details required to support foreign languages and high quality typography. An application programmer using Line Layout and following a few simple guidelines should be able to create foreign versions of an application with little or no code changes.

Also, the toolbox internally uses Line Layout to display text in everything from dialogs to window headers. The standard text editor is also written on top of Line Layout. Applications that implement their own layout algorithms risk having a different and possibly incompatible human interface.

2. Where Line Layout Fits into the System

Line Layout sits on top of the Graphics and Font system, and on top of the low level toolbox. These systems have no textual interfaces and are not dependent upon Line Layout. The following is a brief description of the pieces of the system that fit together to get text to display.

Graphic System (Albert)

Line Layout sits on top of the graphic system. Communication is one way, Line Layout instructs the graphic system to draw or highlight objects. Line Layout does not use the graphic system to obtain character information, including hit-testing information[1]—Line Layout does its own hit-testing.

Low-level Toolbox

For the most part, the low-level toolbox covers those objects that lack a human interface. Line Layout depends directly upon the Styled Text Classes and the Collection Classes.

Fonts

Line Layout works closely with the font mechanism to decide the look and position of characters. Graphic system devices including printers and display also work with fonts to display or print text.

High-Level Toolbox

Any toolbox object that draws text does it through Line Layout. This includes menus, title bars, the standard text editor (ZZText) and all other utilities with a textual human interface.

High-Level Tool Box : includes menus, window titles and a text editor.

F O N T S

Line Layout

Low-Level ToolBox:
includes styled text and collection classes.

Graphics System

Figure 2.1

---

1 Hit-testing is the process of mapping a mouse or cursor position to an object. Normally after an object is 'hit', information is visually passed back to the user indicating that the object can be manipulated.

3. Important Concepts

This section deals with concepts that are fundamental to Line Layout and to this document. Appendix A contains a glossary of other concepts that are also important but are more commonly understood and are not as fundamental to the understanding of this document.

Characters, Glyphs and Unicodes

A character is an abstract object having a single and unique semantic or phonetic meaning. Glyphs represent graphical appearance of characters. For example, the glyphs A, *A*, 𝔸 represent the character A.

Fonts contain a set of glyphs for a particular character or characters. Different fonts may have different glyphs for the same characters. The determination of a glyph always depends upon a font and character context. An application need not worry about the details of obtaining glyphs from a font. Line Layout transparently handles the details of mapping characters to glyphs.

Glyphs do not have a 1-to-1 relationship to characters: depending on the font, a given character may be represented by one or more glyphs (i could be represented by ı plus ·), and two or more characters can display as a single glyph (f and i could be represented by ﬁ). The context of a character also affects the glyph used to represent that character. For example, in cursive font a character may have at least four different glyphs: a separate glyph for the character at the beginning, middle and end of a word; and a glyph for the character in isolation.

Unicode is the name for the 16-bit character code ordering supported by all toolbox utilities and fonts. Unicode does not code glyphs. Glyph coding is the sole responsibility of a font designer.

Fonts

A font is a collection of glyphs that generally have some element of consistency in their appearances (e.g. serifs, or stroke thickness). Fonts also contain other information such as which glyphs represent ligatures, or contextual forms. Within a font, each glyph is associated with a 16-bit code called its glyph ID.

A complex font is a font that contains information associating some glyph IDs with certain combinations of characters and rules. For example, there may be some information in a font that associates the glyph ID $1A01 (which happens to have the appearance 'ﬁ') with the combination of the two characters $0066 ("lower-case f" semantic) followed by $0069 ("lower-case i" semantic). Any font that contains no such associative information is called a simple font. The glyphs in a complex font divide into two classes: rendering forms, for which combination rules appear; and character glyphs, which have a one-to-one correspondence with character codes. Rendering forms include ligatures, applied marks, and contextual forms.

Fonts provide Line Layout with the information needed to make typographic and script formatting decisions without a user's explicit involvement. This contrasts with traditional systems where a user must make an explicit decision on a font by font basis—as a result users settle for low quality output because the burden is too great.

The Bass outline font documentation provides more information about the structure of fonts.

Styled Text

Line Layout makes heavy use of Styled Text objects. The application programmer must understand how to use Styled Text before he can use Line Layout.

The styled text object is the collection of characters (also known as the backing store) that comprise the text for display by Line Layout. All editing of text object is an application's responsibility. An application is also responsible for passing a text object to Line Layout for display. The characters in a text object should always be phonetically ordered as appropriate for the language represented by the characters. Line Layout will display characters in their correct visual order if the characters are phonetically ordered—the next section shows why the two orders are different.

A **text offset** (or character offset) is an index *between* characters in a text object. A text index is the index of a character in a text object. This distinction is important for functions that occur between characters rather than to characters. For example a cursor is a pointer between characters.

Reordering of Characters For Display

A text object is an ordered array of character codes. Character codes are phonetically ordered. Phonetic ordering means that character codes are in their spoken order rather than their written order[2]. The order of character codes within a text object may differ from the written order of the associated glyphs. In particular, Line Layout expects Arabic or Hebrew strings in their spoken order and not in their visually reversed order. The characters in a string of mixed Arabic and English should be in the same order they would be spoken in. This point is very important and bears repeating: Line Layout expects text in phonetic backing store order. Line Layout will reorder glyphs as needed for display purposes. Figure 3.1 shows how Line Layout displays Arabic mixed with English.

As a consequence of reordering characters for display, ambiguities can occur when mapping between text offsets (or insertion points) and the visual location of text offsets. These ambiguities can cause Line Layout to compute multiple carets for a single text offset. Ambiguities also occur independently of a script. For example when a user selects before the first character on a line, a caret can be displayed either after the previous line or at the start of the current line.

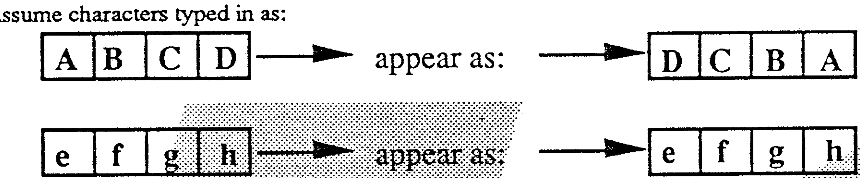
Appendix A provides more detail about character reordering and bi-directional text.

2 This phonetic ordering facilitates many processes upon a text object including text input and parsing.

Character Ordering vs Line Direction Script Ordering

Assume that characters typed in as **ABCD** appear as **DCBA**—as characters might in a script that flows from right-to-left. Also assume that the characters **efqh** appear as **efgh**—as they normally do in a Roman script. The ordering of these characters within a script is refered to as the character ordering.

Character Ordering



A problem occurs when scripts which flow in opposite directions appear on the same line. In this case a line containing the text **ABCDefgh** may display as either **DCBAefgh** or **efghDCBA**. The correct ordering depends upon what the user intended. This ambiguity is resolved in line layout by adding a line direction variant, right-to-left and left-to-right , to horizontal lines. Vertical text does not need this distinction for ordering scripts within a line (it does however need to rotate glyphs within certain scripts.) Line Layout also handles complications cause by numbers, spaces and punctuation that can occur between scripts.

Line Direction Script Ordering

Characters typed in as:

A	B	C	D	e	f	g	h

appear in a left-to-right line as:

D	C	B	A	e	f	g	h

appear in a right to left line as:

e	f	g	h	D	C	B	A

Highlighing the contiguous characters **CDef** in the text string yields the logical, but strange and correct results shown below. This is known as discontinuous highlighting.

D	C	B	A	e	f	g	h

e	f	g	h	D	C	B	A

Figure 3.1

4. Features of Line Layout.

This section describes the various Line Layout features that are visible to an end user. The major part of Line Layout's work is the construction of layouts that describe the appearance of a line of text. To construct a layout, Line Layout needs not only a source of text, but also style information and other options that affect a line's appearance. Four classes of features act together to transform the appearance characters into glyphs, and act to position glyphs. This section list these features by their class type. These class types are: Graphical Features, Layout Features that Affect Character to Glyph Mappings, Layout Features that Affect Glyph Positions and Glyph Adornments.

One important thing to keep in mind: the layout features described below are mandatory for international text, which has many attributes that low-quality English text never needs. While these features are optional for English, they are vital to the correct rendering of text in many other languages.

The human interface for specifying these features are an application's responsibility. An individual application can add additional (non-standard) features. Listed below are "standard" features offered by Line Layout.

Appendix A provides additional detailed information for many features listed here. The actual system styles and method required to control these features can be found in the section System Defined Styles—this is of interest only to application programmers who need to implement a human interface to control the features presented in this section.

Graphical Features

The graphic and font engine supports the following features, which are accessed through Line Layout:

- Text Color

  Text can be styled with any color.

- Text Size

  A User can set text in any point size, including fractional sizes.

The graphic feature set will be expanded to include outlined characters, bold characters, all your favorite Macintosh character styles, and more.

Layout Features that Affect Glyph Mappings

The following features affect character code to glyph code mappings:

- Font

  Text specifies a font for display. If a font is not available, appropriate defaults specified by the application or the system will be chosen. Fonts are specified by their name rather than by an ID number as on the Macintosh. Outlining and shadow effects are a function of the graphic system and are currently only specifiable as a separate font.

Line Layout will display a missing glyph symbol for a character when a font does not have a glyph for that character. In the future we may provide an alternate user interface for missing characters; one where we try to find missing characters by dynamically scanning a font hierarchy until a missing character is found. However, Line Layout would need support from the font cache and/or font manager before such an interface could be implemented. An editor should use a similar font substitution mechanism to style text appropriately. This would allow a user to type any symbol or character without switching fonts.

- Ligatures

  Fonts can instruct Line Layout to display sets of characters as a single glyph. For example, an 'A' followed by an '^' in some fonts will display as an 'Â'. More commonly fonts display 'f' and 'i', and 'f' and 'l' with ligatures, but character pairs like these are best display using contextual forms (described below.)

- Applied Marks

  Fonts can piece together a glyph from multiple glyphs to display a single character. Ligatures and attachments appear the same to a user. The difference is in the technology used to display the character. Unlike ligatures, characters displayed with attachments are manufactured on the fly and do not suffer from the combinatorial limitations of ligatures.

- Contextual Forms

  Contextual forms allow specific character pairs from a font to display using alternate characters. For example, if 'f' and 'i', are displayed using contextual form pairs, a font can instruct Line Layout to display the 'i' without the dot when only the 'i' is preceded by an 'f'. A user need not be concerned whether the contextual form exists; the font will provide the correct result in either case.

  Current Macintosh users must manually type a contextual form, for example option-shift 5 or 6, and manually verify that the contextual form is in the font. If the character is missing, the user must delete the character, already typed-in, and retype the separate parts of the contextual form. Additionally, a user is unable to search the text for a partial part of the contextual form—ie. a user cannot search for an 'i' in a 'fi' contextual form. Line Layout does not have either of these restrictions.

- Reordering

  Line Layout automatically reorders characters, per a font's instructions, for characters in Indic languages. For example, a word typed in as 'hello' may display as 'ehllo' in some scripts. Current systems force users to type the characters according to their visual appearance as opposed to their spelling.

Layout Features that Affect Glyph Positions

The following features affect the placement of glyphs:

- Super/Subscript Positions

  Superscripts and subscripts are relative to the baseline and proportional to the font size. For vertical text the baseline is usually down the center of characters, and super/subscripts are to the right and left of the baseline.

- Letterspacing

  Letterspacing is for moving pairs of characters closer together or farther apart. The amount of movement is proportional to the font size of the characters that contain a letterspacing style. Letterspacing is also known as manual kerning.

- Kerning

  Kerning automatically shift pairs of characters closer together or farther apart. The pairs can shift in both the horizontal and vertical directions. The behavior of the shifting is identical with letterspacing and super/subscripts—the difference being the kerning amount is from a font and the letterspacing and super/subscript amount is from a user.

- Tracking

  Character widths can be expanded or contracted by applying a tracking value to a character. This value is a multiplier applied to the width of characters. A font, depending upon the point size of the text, will scale glyph widths using the multiplier. Tracking causes characters to appear e x p a n d e d or condensed

- White Space Trimming

  This is the ability to ignore white space for purposes of display or measurement on either end of a line. White space trimming allows space characters to hang into a margin. In a Roman script, spaces on the right side of a line are trimmed, this is mandatory if the line is right flush. When a line is centered spaces on both sides of a line are trimmed.

- Justification

  Justification is the ability to make text appear left and right flush simultaneously. Line Layout preforms justification by distributing extra space to spaces and certain classes of other characters in a hierarchical approach. With this approach extra space first goes to white space characters until a user specifiable threshold is reached.

  Line Layout also can add extension bars between characters in cursive scripts, as specified by a font. This functionality is known as Kashida or Insertion Justification—a Kashida is the name of the extension bar commonly used in Arabic scripts for justification.

- Overhanging punctuation/Optical alignment

  These features allow characters specified by a font to 'hang' into the margins. Optical alignment gives the illusion of characters being flush with the margin while they really hang over the margin. Overhanging punctuation such as quotes allows punctuation to hang into the margins. This allows text to appear flush left or right from line to line, with or without punctuation.

- Character Direction

  Character direction allows characters typed in as ABC to be displayed as CBA. A character's Unicode determines its character direction. A user can override a character's default direction.

- Run Direction

  Run direction controls the relative ordering of a blocks of characters. A block or run is defined as a sequence of characters all with the same Character Direction. A character's Unicode determines its run direction. Line Layout uses Run direction when mixed scripts such as Arabic and English are on a single line. Users can override a character's default Run direction—this is necessary in Arabic and Hebrew because sometimes Line Layout cannot format a line as a user wishes because the correct formatting is ambiguous.

Glyph Adornments

An adornment is a graphic applied over a run of glyphs. Adornments do not affect the placement of characters. Adornments are extensible—ie, new adornment styles can be added to text by the addition of a new adornment style by an application. This following lists the standard set of adornments supplied by Line Layout:

- Underline

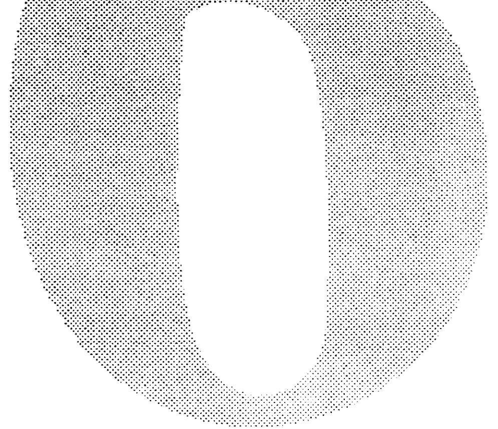
  Single and double underlines can be specified for text. As an option, underlines can be set to ignore spaces and can be set to draw about a line's baseline, or optionally to the baseline of the glyph—which is useful for glyphs that are in a super/subscript position. The width and position of underlines are separately specifiable as is their color.

- Strikethrough

  Same functionality as a single underline.

- Overbar

  Same functionality as a single underline.



5. Objects for Displaying and Hit-testing Text

The following is the set of objects needed by an application to display, highlight, and hit-test text. Two of these objects, TCaretPlace and TInsertionPoint, are used only for hit-testing.

TStyledText

As stated before, a styled text object is used by Line Layout as a source of characters for display. The Style runs in a styled text object also control many of Line Layout's features.

TCaretPlace

A TCaretPlace is a description of where a caret[3] should be placed and the suggested slope of a caret. The actual appearance of a caret is the responsibility of an application. Carets are obtained in pairs from a TLayout. The pair of carets are usually equal, and if so, one can be ignored. However in certain circumstances, when split carets are necessary, an application must use the two carets separately—See Appendix A.

TInsertionPoint

A TInsertionPoint is a way of specifying a position between characters. A TInsertionPoint specifies a TextIndex[4] and a character side[5]. Another way to specify a position between characters is by using a TextOffset—a TextOffset is just an index between characters. Line Layout uses a TInsertionPoint to map between a visual point between characters and a TextOffset—this mapping is known as hit-testing.

Line Layout also uses a TInsertionPoint to figure out the next or previous caret position. Applications need to use Line Layout facilities to compute next and previous TInsertionPoints because a caret position depends upon the visual ordering of glyphs. Applications do not have direct access the glyphs orders except through Line Layout interfaces. In the highly unlikely event that an application wants the ordering of glyphs (and the characters that back them), an application can enumerate through the ordering using TInsertionPoints and a TLayout.

TLayout

TLayout is the object that computes glyphs from characters and then positions glyphs. A TLayout can format, display, highlight, and hit-test text. A TLayout also handles typographic and script specific contextual formatting constraints that are either specified with the characters, determined from a font, or defaulted by the system.

---

3 A caret is a graphical indicator between characters in a stream of text. On the Macintosh a text caret appears as a blinking vertical bar. It is also known as the 'insertion point'.

4 A TextIndex is a zero based 'array index' into a text class used to extract a character at the index.

5 A character side indicates if a hit ocurred before or after a character.

6. How To Use Line Layout

This section describes how to use a TLayout object to display, highlight and hit-test characters. Note that in future versions of this documnet more examples will be added.

Setting Up a TLayout

Through the TLayout::SetText call, an application provides a TLayout with styled text, a style set of line specific information, justification information, white space trimming choices and an origin.

A TLayout operates on a TStyledText object. Normally text will have a font, point size and maybe language styles specified. The complete set of possible styles are documented in the separate section System Defined Styles—most of these styles are of little interest to the average user. If text lacks any particular style, Line Layout will choose a system—or application—specific default. It is important to note that these defaults can vary from system to system, and from application to application. Certain applications need to known when defaults are different because the defaults affect linebreaks, glyph positions and glyph looks. Applications will need to handle the differing defaults in a graceful way.

A TLayout assumes a default origin of (0,0) if an origin is not specified. Providing an origin makes it easier for an application to deal with things like mouse position mappings when hit-testing and the repositioning of lines. Repositioning a line by changing the origin is more convenient for application programs than modifying a TGrafPort's origin—although an application can still reposition a line using the latter method.

An application should provide a Line direction (vertical, horizontal left-to-right, or horizontal right-to-left) as part of a larger set of line styles to a TLayout. A TLayout will assume the default direction of horizontal left-to-right if a direction is not specified.

The line direction style is part of the line style set passed to TLayout::SetText. All default text styles are searched for first in the line style set before TLayout uses a system default. This allows applications to have different defaults from the system default. Applications can choose to have a unique line style object per paragraph for paragraph style defaults.

The TLayout::SetText call includes separate boolean parameters for controlling white space trimming. In the future these booleans maybe specified with the style that specifies line direction, or perhaps as a separate style.

Displaying text

To draw a line of text an application must create a TLayout using the parameters described above. Then, all an application has to do is call TLayout::Draw and pass a port. For replicating a string, it is very fast to redraw the string by changing the origin on a TLayout and calling TLayout::Draw. This is faster than calling TLayout::SetText multiple times.

```
EXAMPLE:      The code below can be used to display an array of Unicodes in
              24 point Helvetica.

#ifndef __LINELAYOUT__
# include <LineLayout.h>
#endif

TLayout layout;
```

```
TFillBundle white(0xffff, 0xffff, 0xffff);

void
DrawString(TGrafPort* port, TGPoint &ppoint, UniChar* text, unsigned long len)
        //Create the styled text object.
        TStyledText styledText(text, len)
        TTextSelection textrange;

        //Style the text
        textrange.SetRange((unsigned long)0, styledText.Length());
        TPointSizeStyle size(24);
        TFontStyle font("Helvetica");
        TStyleSet styles;
        styles.AddStyleToSet(&font);
        styles.AddStyleToSet(&size);
        styledText.SetStyleInRange(&styles, range);

        //initialize the layout with the text.
        layout.SetText(styledText, ppoint);

        //White out the area under the text!
        TGRect retBBox;
        layout.BoundingBox(retBBox);
        port->Fill(retBBox, &white);

        layout.Draw(port);
};
```

It is important to note that any TLayout method that renders to the graphic system may change the state of the TGrafPort. Either an application must save and reset a TGrafPort's state when necessary, or an application must never assume that the graph port's defaults are valid after a TLayout::Draw or TLayout::Highlight call.

Highlighting

Highlighting a line is slightly more complex than displaying a line. To highlight text an application needs to specify a character range and a highlight color. A pair of TextOffsets specify a character range.

TextOffsets and TInsertionPoints are usually computed through the process of hit-testing—hit-testing is described in the next section. Highlighting does not draw glyphs; an application should draw glyphs before highlighting glyphs. To prevent flickering when a line must be redrawn we recommend double buffering. Double buffering is a graphic technique where entire objects are displayed into an offscreen bitmap and then the bitmap is displayed to the screen—without clearing the screen first.

Note that re-highlighting an area to give the affect of de-highlight glyphs will not work because highlighting is not always an invertible function. When deselecting text, an application must redraw a line and re-highlight glyphs as needed.

Note that Discontinuous highlighting can occur when a user selects a partial line of text when the line has glyphs that flow in opposite character directions—as is normal in Arabic. This will happen when the selection is of a contiguous range of characters in the text backing and when the glyphs associated with the characters are discontinuously ordered on the display. Using the normal point/shift-extend algorithms an application will get discontinuous highlighting free.

The next section describes an interface that applications can use to give the appearence of contiguous highlightings when a user selects text with a mouse—as opposed to using a find command to select text.

Hit-testing and Carets

TLayout provides methods to hit-test glyphs and determine the TInsertionPoints associated with a "hit" glyph. A TLayout also provides methods used to determine the number and look of carets associated with a TInsertionPoint. The following is a description of how an application should use a TLayout to hit-test and draw carets.

To map between a TGPoint (usually computed from a mouse position) and a TInsertionPoint, an application should call TLayout::InsertionPointFromPosition. An application can then determine where in the TStyledText object a user selected. To extend a selection an application should compute a pair of TInsertionPoints, one insertion point for the start and end of the selection. An application can easily transform a pair of TInsertionPoints into a text selection for editing and highlighting.

TextRangeFromPosition returns the range of backing store in the StyledText object associated with the glyph at a particular position. This functionality is similar to TLayout::InsertionPoint-FromPosition but may be easier to use for certain applications. For example using this interface an application can determine that a hit on the glyph 'Â' maps onto two separate characters in the text backing. Few applications will need this functionality, unless of course an application needs to highlight a single glyph through a mouse click.

TLayout supplies the methods TLayout::NextInsertionPoint and PreviousInsertionPoint to compute a new TInsertionPoint for a cursor key implementation. Given a TInsertionPoint obtained from InsertionPointFromPosition, these routines return the visual next/previous TInsertionPoint. Failure to use the routines for cursor keys can result in carets jumping around within a line or carets moving in the opposite direction of the arrow on the cursor key[6]. It is also an application's responsibility to handle end cases when a cursor needs to jump to the next or previous line.

An application can use TLayout::CaretPositions to find the caret or carets associated with a TextOffset or TInsertionPoint—the TCaretPlace object contains the position and look information needed by an application. Also the routine below allows applications to determine quickly the slope of the mouse IBeam pointer as it passes over slanted text.

> GetCaretAngleAreaFromPosition[7]—This routine is for changing the angle of the mouse "IBeam" pointer to match the slope of text it is over. This user interface makes it easier for clients to select italic text.

Contiguous highlighting of discontinuous text is a feature where an application can give the user the appearance that text selected with a mouse is contiguous though its backing may really be discontinuous. Users seem to prefer this interface as opposed to showing a discontinuous selection. A TLayout will provide the additional routine below to help applications implement contiguous highlighting if they so wish.

---

6 Currently these bad behaviors can be observed on the Macintosh with pre-system 6.0.7 TextEdit in Arabic or Hebrew.

7 Routines indented in this document are not available yet and are only proposed interfaces.

TextRangeListBetweenTextOffsets—This routine computes the discontinuous text selection associated with the positions of two TextOffset on a line.

Measurement

Different parts of an application are interested in different types of line measurements. What follows is a description of the different measurments available through a TLayout.

To find the minimum rectangle required to display a line, an application should use the method TLayout::BoundingBox. A BoundingBox is useful to determine what will need to be updated if a line is redrawn. A bounding box always includes all trimmed white space on either end of a line.

To measure text for line breaks, an application should turn off justification and leave white spaces "untrimmed." An application can then compute a line break by creating a TLayout for an entire paragraph and using the method TLayout::DistanceBetweenTextOffsets (with possible word break points[8]) to add up the widths of words until they no longer fit on a line.

Once a paragraph has decided the text that belongs on a line, the line may need to be positioned. An application will usually need to make a line right flush, left flush, centered or it must "pin" a specific character on a line to a position.

An application should call the TLayout::Width (or TLayout::Height for vertical text) method to find a line's width to make lines flush left, right, or centered. An application needs to make sure that the leading and trailing white space parameters are set in the TLayout::SetText call. The correct setting is application dependent but normally is a function of line direction and flushness. Also, any ignored white space will not part of a line's width. In addition, leading white space that is ignored will display to the right of the origin.

To help align a particular glyph to a position, use the following methods. Using these routines, an application can compute the amount it needs to offset a line to align a glyph to a location.

GlyphInfoFromTextIndex—This routine will return position information for the glyph associated with a TextIndex. This information will include the glyph origin and bounding box.

TextRangeFromTextIndex—Similar to TextRangeFromPosition. Given a text index, a text range is computed. This routine can be used to compute the Unicodes that clump to form a single glyph. By taking the returned text range, insertion points can be computed and an application can use those insertion points for positioning the single glyph associated with the text range at some point. Applications need this functionality to align text to a decimal tab stop. An application can also use this routine to implement backspace code that backspaces over glyphs instead of characters.

A TLayout assumes that the text between tab stops is a discrete unit. Applications must handle the positioning of text at tab stops themselves. To do this an application needs to use a separate TLayout for each segment of text between each tab stop and a line's endpoints. Then the application must position each TLayout segment independently. The computations for positioning the segments are similar to the computations to position an entire line.

---

8 Linebreaks possibilities can be determined using a the TWordBreak class and a hyphenation dictionary.

Finally the method TLayout::GlyphInfoFromPosition is for special purpose use to allow an embedded graphic within text to take the mouse tracker when the embedded graphic is selected. The section on extending the system will talk more about this. For most applications this method may be of little use.

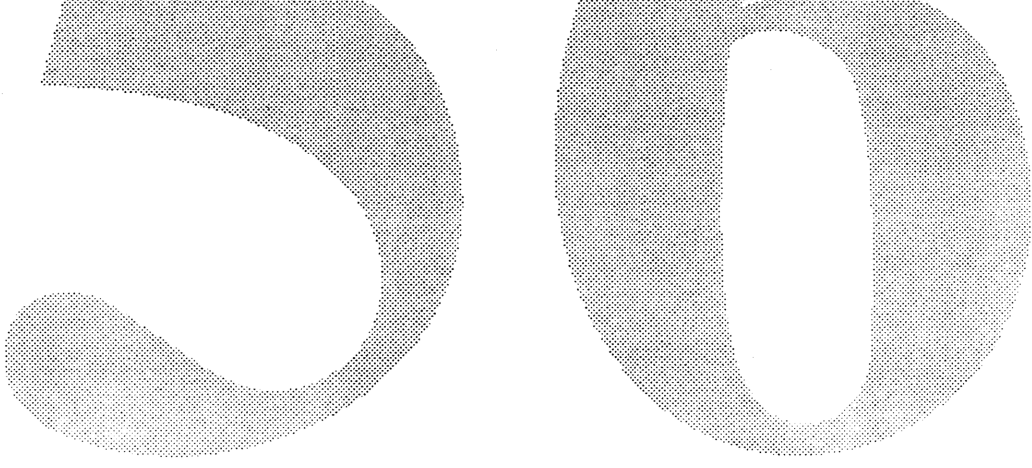
Justification

A TLayout is passed justification information through the SetText interface. A target width and default justification parameters can be specified in the SetText call. The target width does not include trimmed white space on either end of the line. The remainder of the justification parameters come from a justification style in the styled text object and are not final yet. However these parameters will control how much space can stretch, if Kashidas should be inserted, and maximum amounts other characters can stretch.

The current implementation is very simple. A TLayout stretches spaces up to three times their normal width and evenly distributes the remaining space among all the characters on the line. This will change.

Line Breaking

See Appendix D. This section will be filled once some architectural issues are resolved.



7. System Defined Styles

Line Layout directly implements the styles listed in this section. Other text styles not listed here include styles for line height, paragraph line breaking parameters for starters and are implemented by higher level system objects.

Please note that this section is not complete because all styles are not yet defined.

Graphical Styles:

- Text Color

  `TTextColorStyle` is the style used for setting color on text. In the future this will be replaced with a style common to text and graphics—as soon as graphic objects can handle styles.

- Text Size

  `TPointSizeStyle` controls the point size. Size is always in points (72.27 to the inch) and fractional sizes are O.K.

Layout Styles that Affect Glyph Determination:

- Font

  TFontStyle specifies the font for some text. Still undecided is how an application will specify "applied" styles like bold and outline . For now, applied styles can only be specified by a special font name.

- Ligatures and Contextual Forms

  `TMetamorphsisStyle` controls ligatures and forms. Both features are specified through a feature level—see below. When changing only one piece of TMetamorphsisStyle, an application programmer needs to be especially careful to preserve appropriate portions of the other pieces of TMetamorphsisStyle that may already be applied to the text. This is actually a problem inherent to any "compound" style in which a user may only want to change "part" of the style.

  Ligatures and contextual forms have different levels of effect that a user can choose from:

  - The suppress level means no support (i.e. inhibit the layout feature). This might be harsher than expected: in Arabic, for instance, a value of suppress for the ligature feature would inhibit the formation of the usual lam-alif ligature. This setting should therefore only be used if the user wants to see something approaching the naked, unvarnished text source.

  - The mandatory level means to support only mandatory instances of the particular feature. For example, during rendering of Roman characters, Line Layout might only use anchor points in composing an 'é' for an 'e' followed by a '´' in the text source if the "accent anchoring" feature level is at least mandatory. Roman kerning, on the other hand, might never be mandatory, and therefore a value of normal or higher might be needed to enable kerning.

- The normal level means to support "normal" instances of a feature. In Arabic ligaturing, for example, this level is required to get most of the normally used ligatures (such as "initial lam-mim"), assuming the font supports them. This is also the level that most font manufacturers will use for Roman kerning.

- The optional level means to support any and all instances of a feature that the font supports. For example, an "a" followed by an "e" in English text might cause an "æ" ligature to be used if the ligaturing level is optional. The italic 'st' and 'ck' ligatures are similarly optional. Variant appearances of certain Chinese characters might be selected with a forms feature level of optional.

- The fifth option is default. Associated with each layout feature there exists a global default value. These defaults will usually be "mandatory."

For example, these levels allow a user in English to say "never substitute any ligatures," or "substitute only common ligatures like 'fi'," or "do as much ligaturing as possible, including things like 'æ'." To allow for this, these layout features can be individually enabled or disabled, for one character or a range of characters.

- Applied Marks
  — No interface yet.

Layout Styles that Affect Glyph Positions:

- Character Direction/Run Direction

  `TBiDirectionalStyle` controls a character's direction, either left-to-right or right-to-left. Normally text should not include this style. A user will rarely need to override this style. We recommend that if a user must choose a character and run direction that only two of the possible eight choices be shown to the user. The other choices may be too hard to explain to a user or even a programmer. The two choices are (right-to-left, right-to-left) for text that a user wants force right-to-left, and (left-to-right, left-to-right) for text that a user wants to force left-to-right.

- Super/Subscript and Letterspacing Positions

  `TCharacterShiftStyle` specifies super/subscripts amounts as a percentage of point size.

- Kerning

  — No interface yet.

- Tracking

  — No interface yet.

- White Space Trimming

  — No interface yet.

- Justification

  — No interface yet.

- Overhanging punctuation/Optical alignment

  — No interface yet.

Glyph Adornment Styles.

- Underline

  `TUnderlineStyle` is for specifying an underline, its position relative to a line of characters baseline, its thickness, and its color.

- Strikethrough

  `TStrikeThroughStyle` is for specifying a strikeout line, its position relative to a line of characters baseline, its thickness, and its color.

- Overbar

  — No interface yet.

8. Extending the system

An application can extend Line Layout with application specific features and new style types. These section describes how to do that. However, an application programmer first needs to know about a few more system objects and how TLayouts and these extra objects interact. Extending the system is straightforward once the fundamentals are understood.

This section is not for the casual programmer who uses Line Layout to display and edit text strings. It is for the brave who what to extend line layout to display new object types.

Additional Objects

These following objects are for understanding how a TLayout displays and positions characters.

TFont

A TFont is a collection of glyphs, glyph metrics[9] and tables that helps TLayout map characters to glyphs. Applications can subclass fonts to provide enhanced functionality. A particular subclass of interest to applications is one that can display arbitrary graphic objects that do not come from a "font file."

TLayoutGraphic

A TLayoutGraphic is responsible for the displaying, highlighting, and the adornment of glyphs. A TLayoutGraphic is also responsible for all glyph measurement information, including kerning, super/subscripting and tracking.

TLayoutGraphic gets most of its information about a glyph from a font, but the TLayoutGraphic can and will modify the information. A TLayoutGraphic works with the style mechanism to apply features to text. A TLayoutGraphic always has a single font associated with it.[10] However, two TLayoutGraphics may share the same font. Specifically a TLayout Graphic is associated with a unique set of styles. Two characters with identical styles may share the same TLayoutGraphic, while two characters with at least one different style will have different TLayoutGraphics associated with them.

An application can subclass a TLayoutGraphic and embed complex structures into a line of text. These structures can include pictures, graphics, and buttons. These structures themselves can contain lines of text.

---

9 A glyph metric is information about a glyph's geometry. A glyph's width, height and bounding box is an example of a glyph metric. A glyph itself is a collection of curves, or perhaps a bitmap, that is used to draw itself.

10 A fine point: A TLayoutGraphic actually acts like a font subclass and I would like to actually make it a subclass. However, a restriction preventing a font object to be initialized from another font prevents this.

What TLayout does

A TLayout manages a complex interaction among different objects and object types. A TLayout determines glyphs and their positions by controlling the communication among TStyledText, TStyle, TFont and TLayoutGraphic objects and itself.

A TLayout first handles the "bidirectional" reordering of character codes obtained from a styled text object. The reordered character codes are then broken into style runs and the style runs are then associated with a TLayoutGraphic.

Characters are mapped to font-specific glyph codes using the font associated with the TLayoutGraphic. The mappings can be one-to-one, many-to-one, and many-to-many —the more complex mappings result from ligature and contextual form rules. Character rearrangement, for use by Indic scripts and specified by a font, is done at the same time as the character mappings.

Through the reordering, mapping, and rearrangement processes a TLayout maintains a backward mapping of the final glyphs to the original text backing them. This backward mapping includes the index to the character(s) from the text object used to compute the glyph. This backward mapping is for hit testing where it is necessary to find the character index associated with a glyph.

A TLayout and TLayoutGraphic together determine the position of glyphs. The TLayoutGraphic returns the metrics needed to compute positions to the TLayout. A TLayout then computes a TGPoint for each glyph using a glyph's metrics and appropriate justification rules. The TLayout also computes the caret and highlighting points associated with the glyphs—these points are different from the position of the glyphs.

An application can draw, highlight or hit-test a TLayout after all the glyphs and positions are known. TLayout::SetText is the call that computes the glyphs and positions. To Draw, a TLayout simply enumerates through the line passing contiguous runs of glyphs with the same style set to a TLayoutGraphic. The TLayoutGraphic then draws the glyphs and applies adornments.

Figure 8.1 below shows the interaction among the different objects to display the string "the time is five pm." The only arrows missing from the diagram are the communication between a TLayoutGraphic and the style objects used determine the text color (black) and underline. In fact either the TLayoutGraphic, or the styles themselves can apply color or underlines to the text—the responsibility is negotiated among them.

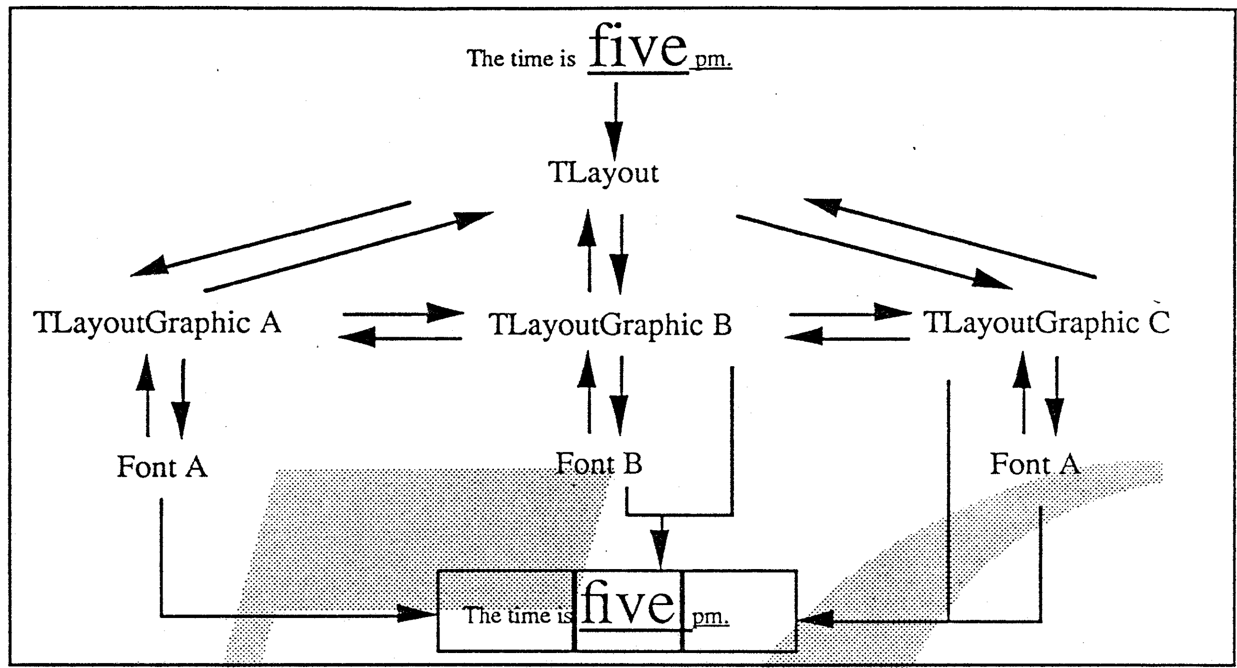


Figure 8.1

Adding New Features

A TLayout can easily accommodate two types of extensions. First, a client may want to add a new type of graphical style to text. For example, a client application may want to put a box around text. Secondly, a client may want to embed other objects in the text such as a button or a graphic. A TLayout is flexible enough to provide both types of extensions.

Ultimately, it would be good if an application only needs to create a single style object to add new functionality. However, many architectural and performance issues must first be solved before this can be done.

Adding New Styles

Adding a new style type, such as boxed text, is easy. To implement a new style an application must create a new TStyle for a box, create a TLayoutGraphic subclass that knows about the box style, and subclass a TLayout to allocate the new TLayoutGraphic subclass.

Subclassing a TLayout to allocate a TLayoutGraphic is done by replacing the AcquireLayoutGraphic and ReleaseAllLayoutGraphics methods. We recommend that an application initialize its TLayoutGraphic subclass from one obtained by calling TLayoutGraphic::AcquireLayoutGraphic.

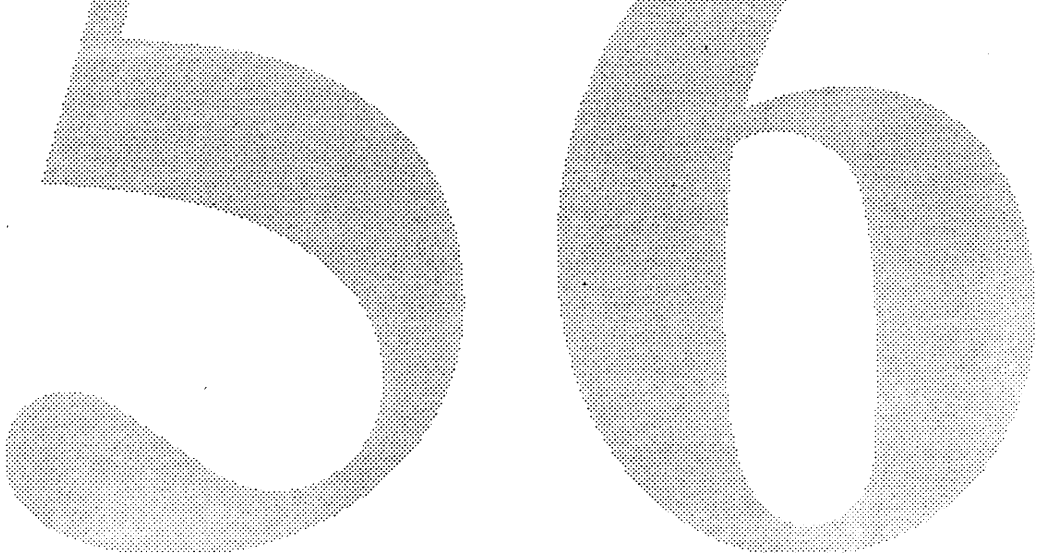
A subclassed TLayoutGraphic must be modified to use the new style type. If the new style will not affect default character metrics then the application is free to modify the DrawGlyphs, HighlightGlyphs, or AdornGlyphs methods as it sees fit. By the way, the AdornGlyphs method is called by DrawGlyphs to add graphical elements to a glyph that do not come from a font—like underlines. If metric information will be changed then the GlyphInfo, and the GlyphAdornInfo

methods need to be modified. The info method must be changes if widths or bounding box information changes as a result of the new style.

Adding Embedded Graphics

Adding a button (or other graphic) to text is not much more complicated than adding a new style. The difference is that a TLayoutGraphic needs to draw a button rather than asking a font to draw a glyph. Also the TLayout::AcquireLayoutGraphic method must be modified to create the appropriate TLayoutGraphic. The TLayout::AcquireLayoutGraphic method can determine if text is text or actually a button by looking at the style associated with the text. Two notes of caution: First, an embedded graphic object or button must be backed by a single Unicode. Second, a NIL TFont should be returned by the TLayout::Font method.

Finally, a button may need to take the mouse when a point down occurs over it. To 'push' the button, the applications should use a tracker object for the line and determine a character hit in the same way it would for any character. The tracker could then tell if the hit occurred on the button by checking the text style associated with the TInsertionPoint obtained from the mouse position. Once it has been determined that the hit occurred on a button, the location of the button can be obtained from the TLayout::GlyphInfoFromPosition method. The application can then hand the mouse tracker to a tracker associated with the button.



9. Performance Issues

Caching of Information By a Client

A TLayout cannot take advantage of information obtained from previous lines. However, an application can. One set of objects that can be reused over and over again are TLayoutGraphic objects.

The method AcquireLayoutGraphic can be used to cache TLayoutGraphic objects between lines. By recognizing that a TLayoutGraphic can be reused from line to line, a client can speed up performance by overriding the AcquireLayoutGraphic and ReleaseAllLayoutGraphic methods. To recognize if a TLayoutGraphic can be reused, an appliction needs to check if the parameters used to construct a TLayoutGraphic remain unchanged from line to line. This kind of optimization cannot be done by a TLayout; it must be done by a client.

Size

A TLayout is a large object; it uses much memory to accomplish its tasks. While an application does not need to worry about the management of this memory, it should avoid using many TLayout objects. If possible, it is recommended that an application use only one object repeatedly. An application should not maintain a separate TLayout for every line that it intends to display.

TLayout Speed

When Line Layout displays a line of text, it takes many machine instructions to get bits on the screen. Some application are time sensitive and need to display text as fast as possible. The speed of Line Layout is a function of the input from an application, and the font used to display the text. A good performance rule of thumb is that the simpler the font and the less styled the text, the faster the text will be displayed. For the fastest performance, text should be in one font style run and the font should only map characters to glyphs—the font should not ligate, kern, or implement contextual forms. Each extra feature or style type applied to the text can cause an extra pass over the text. The style run itself should be simple, it should not kern, it should not superscript, it should not track... The line should not justify, characters should not optically align, and punctuation should not overhang.

Fixed Point instead of GCoordinates

Currently TLayouts work with GCoordinates exclusively. These are big suckers[11] and a layout uses a lot of them. Not only are they big, but they are slow to compute with. We may gain some speed and can reduce the size of a TLayout by using a smaller numeric type. Of course to use a smaller type efficiently, the path of glyph metrics from fonts though the graphic system need to use the smaller type. Otherwise too much time may be spent converting between types.

---

[11] The TGPoint for a single glyph is 20 bytes. A TGPoint is made up of two GCoordinates.

Appendix A— Concepts

This appendix provides a more detailed description, with examples, of some Line Layout features described in Chapter 4. The features described in this appendix are:

- Ligatures, accented forms and accent ligatures,
- applied marks,
- contextual forms,
- automatic kerning,
- automatic cross-stream kerning,
- optical alignment,
- hanging punctuation,
- baselines,
- character reordering,
- split carets, and
- glyph metrics.

Ligatures, accented forms and accent ligatures

A ligature is a rendering form that represents a combination of two or more individual glyphs. The word "ligature" is also used as a verb: to ligature is to form a ligature. Some examples include the 'fi' ligature in English, and the "lam-alif" ligature in Arabic. An accented form may be a special type of ligature: a rendering form that combines a letter with an accent mark. For example, the glyphs 'ä' and 'ò' are accented forms. Multiple accent marks themselves may be present in a font as accent ligatures.

Applied marks

An applied mark is a glyph that is composited dynamically with another glyph. The recipient glyph is called the baseform, and a baseform can have one or more marks applied to it. For example, Polish uses Roman script with some additional glyphs such as ' ś '. If this glyph does not exist as a rendering form in the font, it could be created by dynamically composing the baseform 's' and the applied mark '´'. The composition process will ensure that marks are placed properly with respect to the baseform.

Applied marks are aligned by anchor points. A glyph object associates a base glyph anchor point with each applied mark. Each applied mark has a single distinguished anchor point. This anchor point is lined up with the associated base glyph anchor point for rendering. For example:

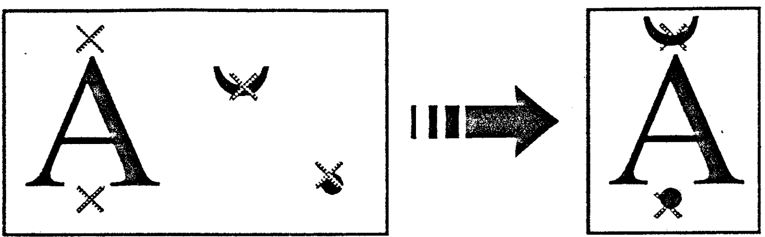


Figure A-1

Contextual forms

A contextual form is an alternate appearance of a glyph that is used in certain contexts. Arabic, for example, has different contextual forms of glyphs, depending on whether they are at the beginning, the middle, or the end of a word. Figure 2 illustrates this by showing the forms of the Arabic letter "ha" that appear alone, at the start, middle, or end of a word. The same character code is stored for each case; it is font's responsibility to choose the correct glyph.

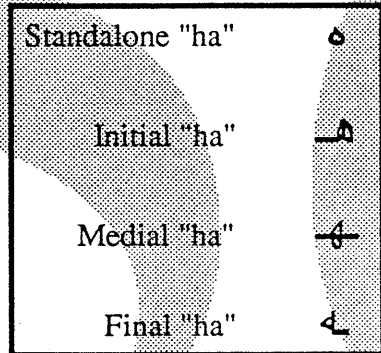


Figure A-2—Arabic contextual forms

A Roman font manufacturer could choose to support either contextual forms or ligatures to present the appearance of a combined 'fi' glyph.

Font has ligature:

f + i => ﬁ

Font has contextual forms:

f + i => f + ì => ﬁ

Figure A-3 — When font sees 'f' followed by 'i'

Automatic Kerning

Kerning is the fine adjustment to the normal spacing that occurs between two or more glyphs, usually to improve the apparent letter spacing between glyphs that "fit together" naturally. Kerning is driven via font tables to determine how much to increase or decrease the space between two glyphs.

To To

Figure A-4 — An 'o' kerned with a 'T'.

Automatic Cross-Stream Kerning

Cross-stream kerning allows the automatic movement of glyphs perpendicular to the line orientation of the text. This means glyphs move vertically in horizontal lines. Cross-stream kerning is driven via font tables to determine how much to shift glyphs. For example, a hyphen between two capital letters should be raised to reflect the centers of those glyphs:

X - Y vs. X - Y

Figure 5 — Cross-Stream Kerning Example.

Optical Alignment

Without additional information, glyphs may seem to line up incorrectly on the right and left margins. This is accounted for by two factors. First, glyph advance-widths contain a certain amount of extra white space to account for the normal inter-glyph spacing.

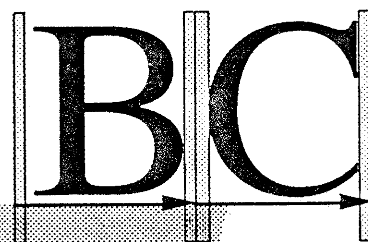


Figure A-5

This produces certain anomalies, since this space varies with the font size. For example, if different sizes of the same font are left (or right) flush in QuickDraw, they will not generally line up correctly:

Figure A-6

The second problem is that due to certain optical effects, curved lines do not appear to line up properly with straight lines. To make them appear to line up, some compensation must occur. For this reason, curved letters such as C or S are generally designed to extend slightly below the baseline, so that they appear to line up with straight letters such as H.

Figure A-7

This same effect happens on the right and left edges of lines. In both of the following example, the center O lines up with the H's, in the sense that the leftmost black edge of the H is even with the leftmost black edge of the O; this is shown by the line on the right. However, the letters do not appear to align correctly because of optical effects, as you see by looking at the words on the left.

Hotel... Hotel...
Other... Other...
Hotel... Hotel...

Figure A-8

To compensate for these effects, fonts contain optical alignment information. When determining the right and left edges of a line of text, whether at tab stops or line starts, Line Layout will use the optical right and optical left edge. [Note that corresponding top and bottom optical edges are available for vertical text as well.]

Hanging Punctuation

Hanging punctuation is punctuation that hangs outside the bounds of the text. This commonly includes very light-weight punctuation such as periods or quotation marks:

"The quick brown fox jumped
over the lazy dog," said the sage.

Figure A-9

A font determines which glyphs overhang and by how much.

Baselines

The baseline of a character is a line that defines the position of the character with respect to other characters. The importance of the baseline is illustrated by different sizes of characters; where it shows a stable point, out from which the characters grow. This is also the case with other scripts: the ascent portion of a character grows upwards from the baseline, while the descent portion of the character grows downwards. However, there can be dramatic differences in the general proportions of characters with reference to the baseline. If we were to take the naïve approach to glyph placement, we would end up with something like figure 10.

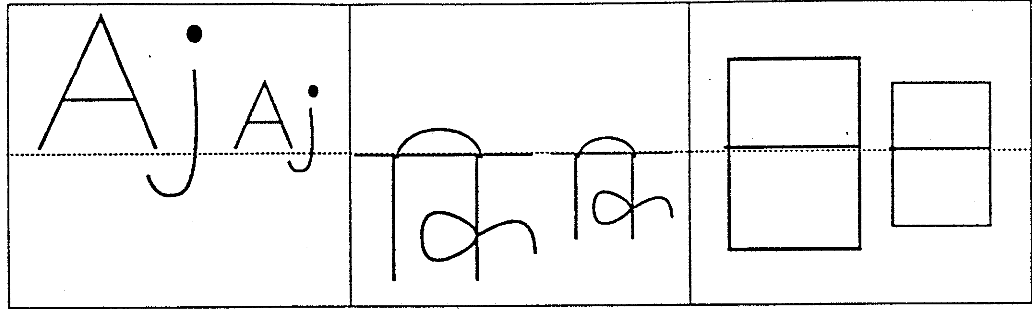


Figure A-10

In this figure, the baselines within a script match correctly, but the relationship between different scripts is incorrect. We need to align the baselines on an inter-script basis, so that we can get something like figure 11. This effect could always be done manually. However, this would require the end user to manually adjust the positioning whenever size adjustments were made within a line, or require the client application to have considerable knowledge about scripts.

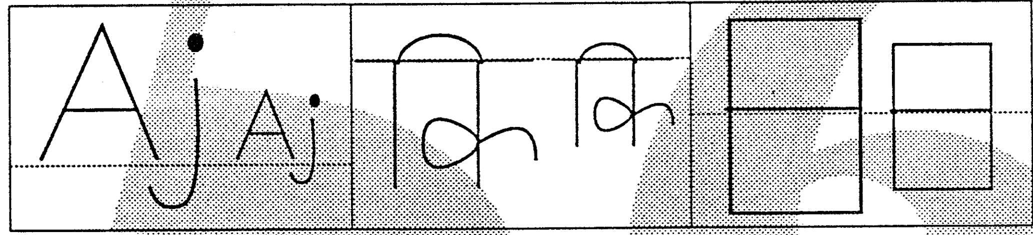


Figure A-11

We have not fully worked out the means of specifying this interscript alignment. One approach is to have fonts identify their natural baseline (Roman, Indic, Centered . . .) and to have an application specify (through a paragraph or line specfic style) the amount to shift characters with respect to a particular baseline. The shift amount could either be dynamically or statically computed by an application—the user interface for this may be tricky.

For example: When Line Layout parses some text it will get a font associated with the text. The font will have a value indicating its natural baseline, lets say Indic for now. Line Layout can then quary a paragraph for an Indic baseline shift rule. A 12 point "Roman" paragraph can then decree that all Indic baselines within the paragraph should shift up 9 points. If a paragraph wanted to produce something like figure 10 it could return 0.

The protocal between Line Layout and a text editor is simple. The hard part is letting the paragraph determine the shift amount. One possible mechanism is for a paragraph to have a user settable default paragraph baseline and a baseline rule. The baseline rule, when asked for a shift amount, can detect if the request is for a baseline other than the default paragraph baseline and can return the appropiate shift amount.

The default baseline rule could compute the shift amount from some 'user settable' template character associated with the line, paragraph, or document. From this template character the baseline rule can associate all possible baselines with a shift amount.

Character Reordering

A given text object may include parts that are rendered in opposite directions. For example, Arabic letters print right-to-left, while Arabic numerals and Roman script print left to right. A direction streak is a series of characters, next to each other in a text object, which belong have the same dominant rendering direction—left-to-right or right-to-left.

A single line of text with streaks running in opposite directions can be formatted in two different ways depending upon the dominant line/paragraph direction. Figure 9 shows the possibilities—in this example line direction is refered to as run succession.

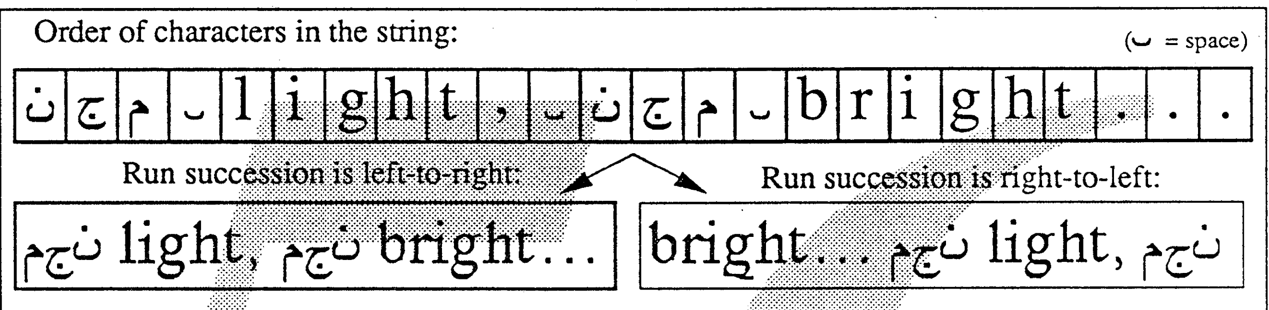


Figure 9

Even within a direction streak, some characters may render in an order that is different than the backing store order. For example, in Devanagari script a short 'i' prints before the consonant that it logically follows: the word "Hindi," when printed in Devanagari, comes out something like "ihndi".

In the above cases, it is assumed that the characters in the string appear in "natural" (i.e. phonetic, semantic) order. Taking the example above, the backing store would contain character codes in the order "hindi," not in the order "ihndi." Line Layout—not the string creator—is responsible for all reorderings. This point cannot be too highly emphasized: as far as Line Layout's clients are concerned, they always present text to Line Layout in phonetic backing store order, and leave the reordering up to Line Layout.

Split Carets

Because of character reordering, ambiguities occur when mapping between text offsets (or insertion points) and the visual location of text offsets. As a result, the user must sometimes be presented with multiple carets. Specifically two carets are needed for right-to-left and for Indic scripts. Figure 10 shows by example why multiple carets are needed for right-to-left scripts. For right-to-left scripts the split carets can suggest where the next left-to-right or right-to-left character will appear.

Multiple Caret Example

Assume that when a user types **ABCD** it appears as **DCBA**. Also assume that when a user types **1234** it appears as **1234**. Now assume that the user types **ABC234** and it appears as **CBA234**.

A	B	C	2	3	4

Order in which characters are typed.

C	B	A	2	3	4

Order in which characters appear.

Now suppose that the user wants to enter a character before the '2', so the user points down on the left side of the '2'. Where should the caret go? That depends upon the next character typed. A '**D**', typed before the '2' will appear to the left of the '**C**'; however a '**1**' will appear between the '**A**' and '2. A solution is to display two carets, one to the left of the 'C', and one between the 'A' and '2'. This example is analogous to the problems encountered in Arabic and Hindi.

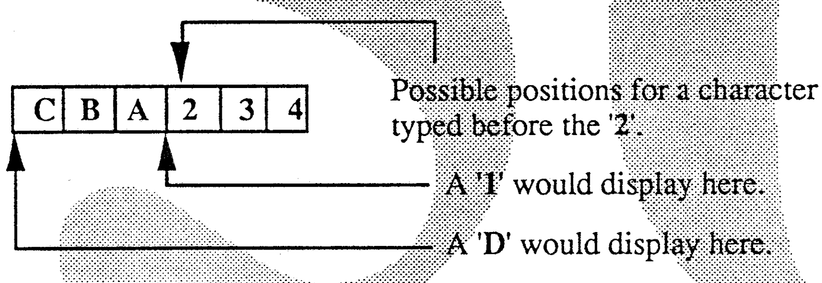


Figure A-14

Because the concept of multiple carets is complicated, an alternate description of the need for multiple carets follows:

> While displaying a line of text, characters/glyphs get sorted into visual positions. In some scripts, this sorting causes the visual order to have little relation to the "typed-in" order. In others words, characters can get scrambled around on the screen when you type, however the system remembers the order in which they were typed.
>
> A text offset points "between" characters in their backing (or typed-in) order. For example: In the word 'Hello', the text offset of 1 is between the 'H' and 'e'. When displaying a caret in this case, only one caret appears between the 'H' and 'e'. Split carets occur when characters that were typed-in next to each other get visually reordered relative to each other when displayed. So for example, if the word 'Hello' displays as 'eHllo', one caret will appear before the 'e' and one caret will appear after the 'H'.

Glyph Metrics

Figure 15 gives an overview of the graphic information that should be supplied by the Font Manager on a per-glyph basis. Some features in the diagram are only important for certain directions of text; the vertical advance width and origin, and the top and bottom optical boundaries are used if the run succession indicates vertical text, while the corresponding horizontal advance width and origin, and the left and right optical boundaries are only used if the run succession indicates horizontal text. Anchor points are only used to place applied marks, while the black bounds are only used for heuristic placement of accents when there is insufficient anchor point information.

A number of these features are optional: if they are not supplied, then the Font Manager will have to make due with the information it has. For example, if the right and left optical boundaries are not included, then they should default to the origin and advance width. If the vertical origin and advance width are not defined, then they will be set to the midpoint of the advance width and the glyph height respectively.

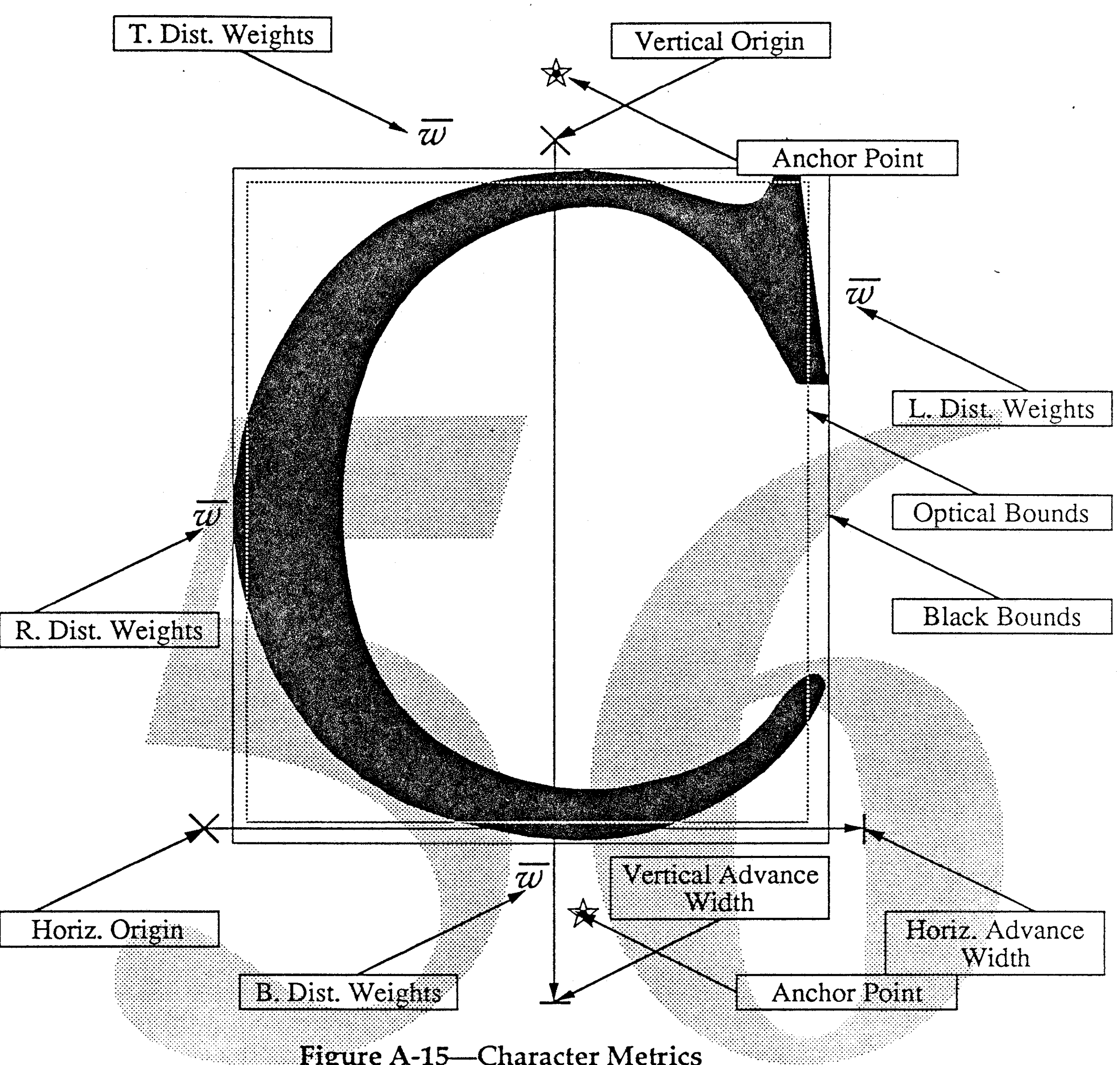


Figure A-15—Character Metrics

Appendix B — Application Design Requirements

Line Layout and the base Text Classes carry out the hard part required to make text international within an application. However, an application needs to implement a few features to make text entirely international. This Appendix lists the minimal requirements an application should follow.

Paragraphs, Tabs, and Rulers.

Applications need to let users choose a paragraph's line direction style and applications need to handle tabs and rulers according to the line direction. The choices for line direction are:

- Horizontal.

  Horizontal text can have one of two possible paragraph line directions, one for left-to-right "Roman-like" paragraphs and one for right-to-left "Arabic-like" paragraphs. The correct line direction choice depends upon the "predominant" script direction for an entire paragraph—paragraphs may contain text with different script directions. A user needs direct control of the line direction setting. An incorrect line direction choice will cause a TLayout to display lines poorly.

  Once a choice is made, an application needs to be sure that the scales for rulers in right-to-left paragraphs flow right-to-left. Also tabs stop positions must flow from right-to-left. This implies that when a user changes a paragraph's line direction, the text between tab-stops gets flipped around. For example: the characters 'ABC' TAB 'DEF' will display as the glyphs 'DEF' TAB 'ABC' in a right-to-left paragraph. It is important to remember that Line Layout only handles text between tab stops and a line's end points, and the application is responsible for handling tabs.

  The rules for placing successive lines of right-to-left text are the same as for English., but special care must be taken to make lines either flush right or flush left. Normally left side white space needs trimming in flush-left right-to-left lines and right side white space needs trimming in flush-right left-to-right lines.

- Vertical.

  A paragraph's line direction is almost always vertical in a vertical column. Text and tabs within a vertical line flow from top-to-botton. Successive lines can flow from either left-to-right or right-to-left. Rulers and tabs need to behave accordingly. Also, vertical lines are almost always top flush.

Multi-Columns Pages.

A text-chain is a contiguous stream of text that is part of the same letter, same story or same article. Examples include: Traditional word processors where the text that flows between columns is in a single chain, page layout programs where the text in a single story is in a single chain, and terminal emulators where the text in the log file is in a single chain.

A text-chain direction determines the placement of columns—for most applications the term column direction can be used for text-chain direction. Applications should flow columns across a

page from right-to-left if its associated text chain is right-to-left and from top-to-bottom on a page if its associated text chain is top-to-bottom.

A text-chain direction can also limit a user's choice of a paragraph's line direction. For example an application likely wants to deal with only horizontal paragraphs in a horizontal chain and only vertical paragraphs in a vertical chain. A text-chain may simultaneously contain paragraphs with right-to-left and left-to-right line direction settings. A single text chain usually does not contain both horizontal and vertical lines.

An application with a single column may not need a text-chain direction, but still might want one to use for defaulting a paragraph's line direction and flushness.

Documents and Page Numbers.

Applications need to let users order successive pages from front to back or back to front, change placement of page numbers, and bind pages on either side. This flexibility is mandatory for some users because it is common with some scripts to bind books 'backwards' from the way most English books are bound.

Also users need to specify page numbers using alternate numbering systems and patterns. The toolbox will provide objects that can generate numbering patterns for any specified language.

Mixed Scripts Within a Paragraph.

Finally, an application must provide support to Line Layout to get the baselines of mixed scripts visually aligned—see Appendix A for more details.

56

Appendix C — Unsatisfied External Requirements

This section lists the areas of the system that depend upon Line Layout, or that Line Layout depends upon which are currently undefined. Until these areas are defined and implemented the promiss of Line Layout will not be fulfilled.

Font Architecture

The entire font mechanism is missing. Below I've listed only the most critical missing portions related to Line Layout.

Missing Fonts and Missing Glyphs

Currently Line Layout uses a system default font when a specific font is missing. However the system default may not be a good match—where a good match is when two different fonts handle the same Unicodes. We need a 'fast' mechanism to find a font that matches well with a font that is missing.

Getting Metrics

The mechanism for getting font metrics is not defined. In particular Line Layout needs a font's slope and the following tables—with both horizontal and vertical variants: attachment tables, kerning tables, glyph property tables, tracking tables, and character to glyph mapping tables (for all swash styles).

We also need quick method for getting a glyph's width for horizontal text; height for vertical text; attachment points, ascent, descent, and bounding box. What we have now does not work. For example we cannot format Japanese text because our font mechanism cannot handle that many glyphs. It is very important to get an implementation that works because we need to test the path of Japanese through the system. This testing will place much more stress on the system than Roman will.

Human Interface

Line Layout provides a lot of functionality but it lacks user interface. The ultimate user interface is application-specific, but we need to provide a standard presentation of the style information used to control Line Layout.

Additionally, we need to investigate the use of contiguous selection of bi-directional text—see the section on highlighting. The implications of contiguous selection is not well understood. We need to look specifically at cutting text, copying text, pasting text, searching text, and replacing text.

Passing Application specific text styles

How should the system handle styled text where some styles may not be understood by other systems? This is the same problem occurs when passing any object to another system. Because we do not pass the methods that operated on the object with the object, we may not know how to use an object. We need a set of standard rules for dealing with unknown objects.

Validation of Line Breaks by an Editor

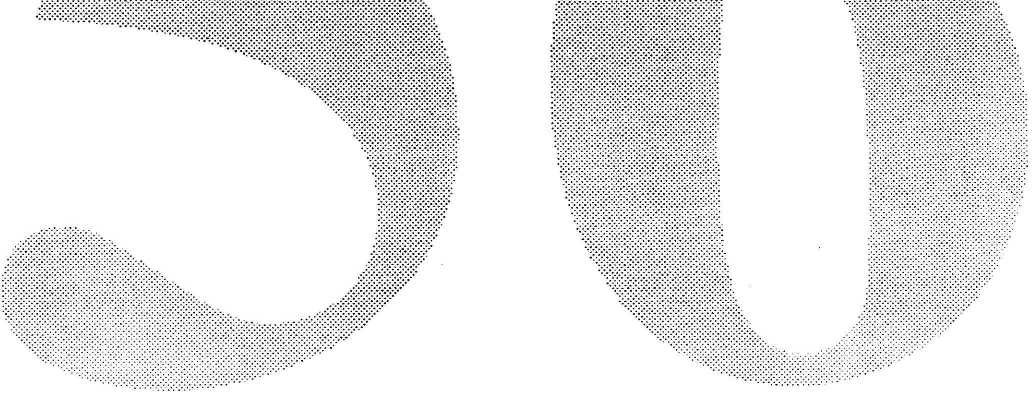
Applications must be sensitive to configuration changes that can force the recomputation of line breaks. For example, font defaults that may differ from system to system will often cause different line breaks for a single document displayed on two different systems. This will usually result in the document paginating differently on both systems. For some users the new line breaks can be unacceptable and an application might have to query a user for advice.

It is worse than described above because many things can affect line breaks. A wide range of dynamic and static configuration changes within a system and between systems can effect line breaks. For example, it can get real sticky when an embedded graphic, within a line of text, is dynamically linked such that when a change occurs the graphic cannot redraw itself before the linebreak is recalculated. Forward reference to a page number is another example.

In the case of forward references to a page number an application may have trouble computing the 'width' of the reference because it is not known and whenthe width is known it can cause a line to rebreak and as a result recursively cause the reference change again.

The collaboration mechanism also will need to address this issue—it is a general problem. For example, on two machines configured differently, how can we present the same image of a document for shared editing? Can it be done and if not how can we check configurations?

We need an architecture for dealing with these problems and just as important we need a standard user interface to allow a user to control what to do when a problem arises.



Appendix D — Open Architectural Issues

Styles

Many standard text styles are undefined. These undefined styles fall into two classes:

The first class of styles are those where the lower level system architecture required to define 'algorithmic' styles do not exist. This set of styles includes graphic and font styles like outline and **bold**. Currently the interface into the graphic and font system for these styles have not been defined. Line Layout will support these styles as soon as they are defined and as soon as the methods for selecting the styles are define.

The second class of styles are those styles where we have not had the time to get around to thinking about the details of the interfaces. Styles for kerning, tracking, and mixed 'language' baselines are an example where we know what we want to do, we know how to do it, but we have not had the time to spend working on the interface.

Simplified Programmatic Style Extensions to Line Layout

It would be nice if an application could add new features to Line Layout entirely through the style mechanisms. In general this can't be done, but we can do a better job with some classes of styles.

For example, it would be nice if we could make adornments easier to implement by only subclassing an adornment style. We have ideas how to do this but it is low on the list of things to do. If we have the time to generalize adornments we will, but for now we have an extension mechanism that works.

Line Breaking

Line Layout needs a more complex method for computing line breaks. The incremental measurement method described in this document, where an application adds the width of a line's pieces, does not always work. One reason the algorithm fails is that the character to glyph mapping indirectly depends on the characters that are on the line.

Also, an application, using an incremental line break algorithm, cannot compute the glyphs which may overhang a line at a particular break point. Glyphs that can overhang include punctuation and trimmed white space. The reason the incremental algorithm can't work is that it assumes glyphs are laid in the same direction as the characters in the text backing. This is not true, and as a result line end calculations may be made using the wrong glyphs.

Finally, hyphenation of words can cause character metric changes and in certain languages cause spelling changes. For example, take the word 'office'. The 'ffi' might ligate into a single glyph. If we break the word into two pieces, like between the two f's because of hyphenation, the width of the two pieces (ignoring the width of the hyphen) may not equal the width of the whole word!

In most cases, even with these failures of an incremental algorithm, the line breaks computed by an application will be close enough. This is because an error will usually cause an application's

calculation to be off by a small amount and the error will usually be on the side of creating shorter rather than longer lines.

To compute an accurate break point, an application must recompute an entire line for each possible break point and pick the line with the 'best' width. The problem with this approach is that this is slow: Currently we do not have a fast and accurate algorithm to solve the problem. More importantly, if we did have a faster algorithm it is not clear how it will affect ZZText's Knuthian line breaking algorithm.

Finally Line Layout currently does not handle soft hyphens at the ends of lines. It lets a font choose to display hyphens or not. Line Layout needs to directly handle hyphens. Line Layout must convert soft hyphens to real Unicode hyphens—or NULL Unicode characters depending on context—before a Unicode to glyph mapping and before character reordering and rearrangement.

Device Dependency

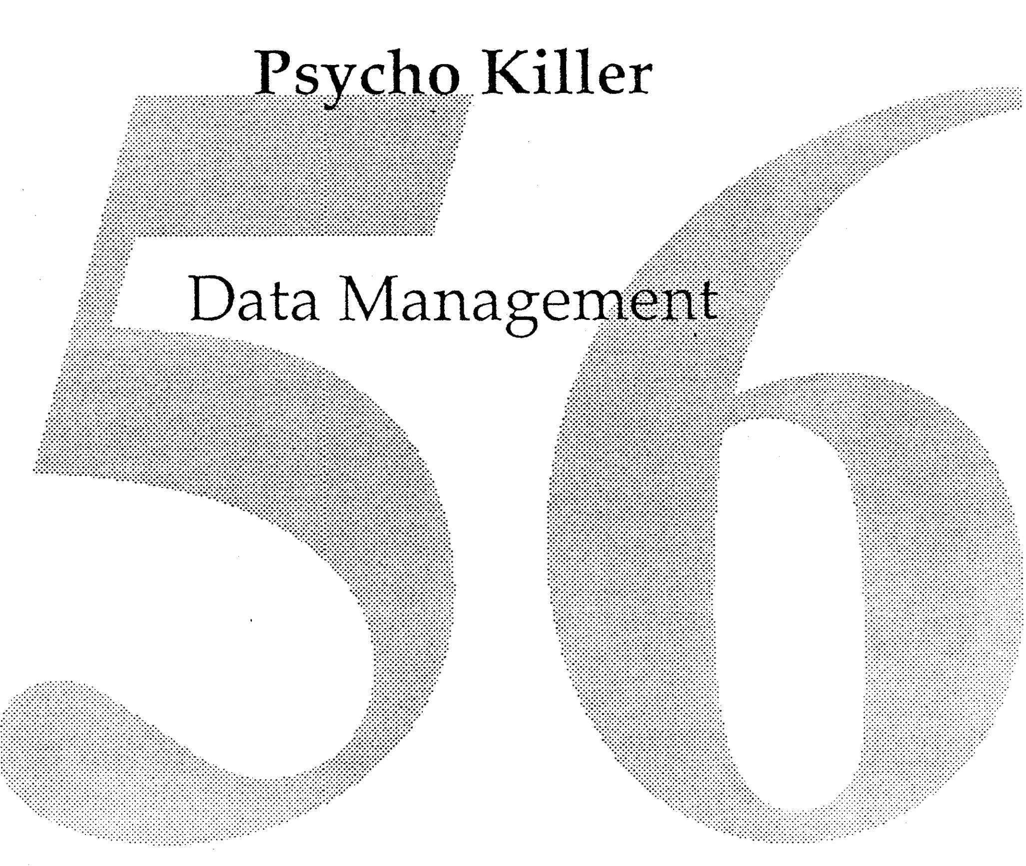
Currently Line Layout feeds the graphics system ideal glyph positions. As a result the spacing between characters appear uneven on low resolution output devices (including screen displays.)

Most systems use some type of device dependent positioning algorithm to improve the spacing of glyphs. For example Display Postscript allows applications to select between device specific and/or device dependent font widths. Applications can optionally compute line breaks using a device dependent or independent method. Applications can then place characters at either device dependent or independent positions—depend upon an application's need.

Currently Pink lacks clear plan to deal with the uneven glyph spacing problem.

56

Psycho Killer



Data Management

56

Psycho Killer

Qu'est-ce que c'est?

Arn Schaeffer x48117

Psycho Killer is the data management project for Pink. This document describes the classes you need to manage data on the disk.

56

Introduction

The Psycho Killer project provides classes which aid in the management of data in the Pink world. Transactions and logging will be fully supported when the Credence classes are available.

Architectural Overview

Psycho Killer provides classes which aid in the management of disk based data in the Pink world. Classes are provided for managing objects on disk (like the Chunk Manager) and for finding (indexing) those objects. Psycho Killer will use the Credence classes to support transactions and logging.

Psycho Killer provides classes which associate a key with a value. In this respect, all of the classes resemble the `TDictionary` class found in the Utility Classes. The simplest of these classes associates a fixed-key with an arbitrary (heterogeneous) value. For example, a class roughly equivalent to the Chunk Manager would return handles from requests to store arbitrary data on the disk. These handles can be used to retrieve the data at a later time. Because the handles are assigned by the system, they can have a one-to-one correspondance to a physical file location where the object can be found. If you are comfortable dealing with handles in this way then this class is absolutely the most efficient class to use. It will always take at most one disk access to find your data given a handle.

Psycho Killer also provides classes for associating arbitrary (homogeneous) keys with arbitrary (heterogeneous) values. Once again, the interface for this kind of class resembles a TDictionary. There is obviously no direct mapping to a file location when the keys can be arbitrary user objects. Because of this, some type of indexing technology must be used in these classes. The first release of pink will include atleast one kind of class that uses an indexing technology to store arbitrary (homogeneous) keys. This class will implement a kind of dynamic hash table. Dynamic hash tables provide O(1) access time to the keys. They allow for enumeration of all of the keys; however, they do not allow for an in-order enumeration of the keys based on some internal (to the keys) notion of ordering. Another appropriate indexing technology to explore as an extension to Psycho Killer would certainly be some kind of btree class.

While the indexing classes will allow you to associate arbitrary keys to arbitrary values, there are many advantages to associating arbitrary keys to fixed-values such as the handles returned from a class like a Chunk Manager. First, using a level of indirection allows multiple indicies to access the same value. This can be valuable in many applications. Second, all indexing technologies have a certain amount of reorganization overhead associated with them. In dynamic hash tables, keys and values are shifted from bucket to bucket as the index grows. In btrees, keys and values are copied to new nodes as nodes are split and collapsed (however, there is much less copying in a btree). The single class discussed in this document uses a dynamic hash table for its indexing technology and a chunk manager like record manager for managing the values. At this time, the interface for the individual parts is still up in the air; however, a usefull class, `TDiskDictionary`, is presented.

Classes

TDiskDictionary[1]

A `TDiskDictionary` is used for the storage of objects on the disk. It works like a `TDictionary`. You can have arbitrary keys and arbitrary values. Objects which are placed in disk dictionaries should override the `IsEqual()` method and the `Hash()` method.[2] The interface presented here was meant as

1. For all the algorithm widget heads out there, a TDiskDictionary uses a directoryless dynamic hash table for its index. It also uses a spiral storage scheme to improve the utilization of all of the buckets and cut down on overflow buckets.
2. It is very important that your `Hash()` method returns hash values which are uniformly distributed over the values that can be represented in a long. If your hash function does not

an approachable (working) first version.

```
class TDiskDictionary : public MCollectible {
   public:
      TDiskDictionary(char* fileName);
      virtual                       TDiskDictionary();
      virtual     void              Add(MCollectible* key, MCollectible* value,
                                       Boolean replace=TRUE);
      virtual     Boolean           Remove(const MCollectible& key);
      virtual     Boolean           Member(const MCollectible& key) const;
      virtual     MCollectible*     Retrieve(const MCollectible& key) const;
      virtual     void              Commit();
      virtual     TIterator*        Iterator();

      virtual     void              SetDestroyFileOnDelete(Boolean doIt = FALSE);
};
```

`TDiskDictionary::TDiskDictionary(char* filename)`
Create a new `TDiskDictionary` object.

`TDiskDictionary::TDiskDictionary()`
Delete the `TDiskDictionary` object from memory. Close the associated files on disk. Uncommitted operations are lost. The associated disk files are only deleted if the destroy file on delete flag is set.

```
void TDiskDictionary::Add(MCollectible* key, MCollectible* value,
                          Boolean replace = TRUE)
```
Add a key, value pair to the disk dictionary. If the replace flag is FALSE and there is a key which "is equal" to the passed in key then do nothing. If the replace flag is TRUE and there is a key which "is equal" to the passed in key then replace the value with the passed in value. If there is no existing entry then add the key, value pair regardless of the replace flag.

`Boolean TDiskDictionary::Remove(const MCollectible& key)`
Remove the key (and its associated value) from the disk dictionary. Return TRUE if the key was actually in the dictionary.

`Boolean TDiskDictionary::Member(const MCollectible& key) const`
Return `TRUE` if this key is in the disk dictionary.

`MCollectible* TDiskDictionary::Retrieve(const MCollectible& key) const`
Retrieve a record from the disk dictionary using the passed-in key. Return the value associated with this key or `NIL` if no value is found.

`void TDiskDictionary::Commit()`
Commit all changes since the last `Commit` operation to the disk dictionary.

`void TDiskDictionary::SetDestroyFileOnDelete(Boolean doIt = FALSE)`
Normally, deleting the disk dictionary object in memory does not actually destroy the data on the disk. Setting the destroy file on delete flag will cause the disk files to be deleted. This routine is not implemented for d11.

`TIterator* TDiskDictionary::Iterator()`
Return an iterator for iterating over all of the objects in the disk dictionary. The values returned from the

---

currently do that, a simple technique is to use the result of your existing hash method as a seed to a `TRandomNumberGenerator` and grab the next random number.

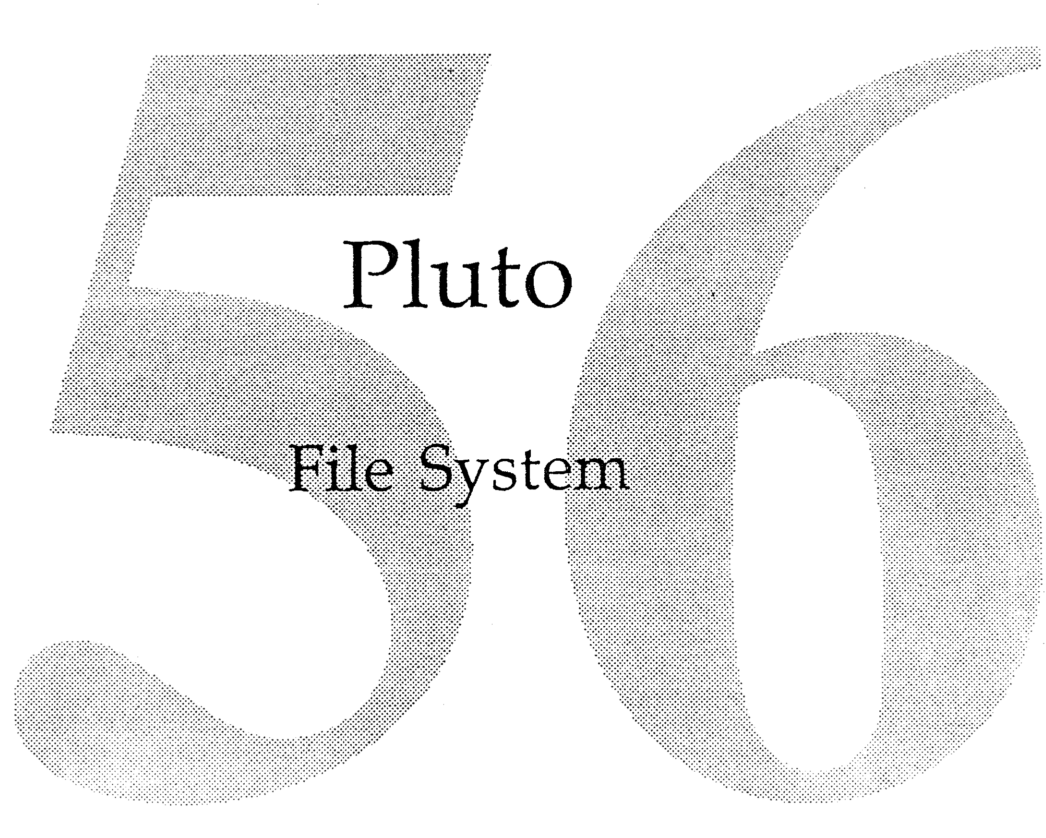
iterator's `Next()` & `First()` routine are `TAssocs` where the `key, value` pair in the `TAssoc` corresponds to the `key, value` pair in the disk dictionary. This routine is not implemented for d11.

Example Code

This is some example code for adding key, value pairs to the disk dictionary and querying the disk dictionary for the values.

```
sampleroutine()
{
   // Create a new disk dictionary.
   TDiskDictionary* myDictionary = new TDiskDictionary("arndb");

   TCollectibleLong key1(11);
   TCollectibleLong value1(1001);

   TCollectibleLong key2(5);
   TCollectibleLong value2(999);

   TCollectibleLong key3(75);
   TCollectibleLong value3(102456);

   // Add some entries to the disk dictionary
   myDictionary->Add(&key1, &value1);
   myDictionary->Add(&key2, &value2);
   myDictionary->Add(&key3, &value3);

   // Commit the changes
   myDictionary->Commit();

   // Delete the object in memory
   delete myDictionary;



   // ...Some time later....
   myDictionary = new TDiskDictionary("arndb");

   // Checking to see if various keys are in the dictionary.
   Boolean found = myDictionary->Member(TCollectibleLong(11));
   qprintf("11 was found as a key in the dictionary = %d\n", found);

   found = myDictionary->Member(TCollectibleLong(1001));
   qprintf("1001 was found as a key in the dictionary = %d\n", found);

   found = myDictionary->Member(TCollectibleLong(5));
   qprintf("5 was found as a key in the dictionary = %d\n", found);

   found = myDictionary->Member(TCollectibleLong(69));
   qprintf("69 was found as a key in the dictionary = %d\n", found);

   // Retrieving a value given a key
   TCollectibleLong* someLong = (TCollectibleLong*)
```

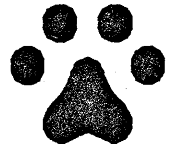
```
                                myDictionary->Retrieve(TCollectibleLong(11));
    qprintf("Value associated with 11 is: %d\n", someLong->GetValue());
    delete someLong;

    someLong = (TCollectibleLong*)
                        myDictionary->Retrieve(TCollectibleLong(75));
    qprintf("Value associated with 75 is: %d\n", someLong->GetValue());

    delete someLong;
    delete myDictionary;
}
```



Pluto

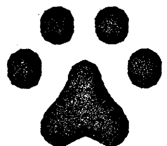
File System

56



Pluto
Architectural Overview

Chris McFall



56

Introduction

Pluto is the file management software for the Pink system. This document presents a discussion of the architecture of Pluto, and of the capabilities that Pluto affords to its clients.

Goals

The goals of the Pluto architecture are to support the functional requirements of the Pink system with respect to file management, and at the same time to maintain a high degree of reliability and a high level of performance. Pink is a highly-leveraged, integrated software system, in which, in order to deliver the best experience to the user, all of the components must work together well. Functionality and reliability requirements are often at odds with performance requirements; so many tradeoffs will no doubt have to be made to achieve a satisfactory balance.

Support Pink

The Pluto services are intended to meet the needs of the Pink Toolbox software and of application programs. Some principal clients of Pluto are the following:

Opus/2. The operating system kernel [1] relies on Pluto for the management of backing store for virtual memory.

Valhalla and Odin. The Pink Finder [2] and the Desktop Manager [3] rely on Pluto for file storage and for file directory service. The desktop presents a filtered view of the directory structures and file system objects stored on the various mounted volumes, and, consequently, Pluto and the Desktop Manager must cooperate to ensure that the view is kept accurate. The Experimental Finder project, now under way, will approach this problem by channeling most file system operations through the Desktop Manager.

Credence. The concurrency-control and recovery software [4] relies on Pluto for a notification service to support its write-ahead logging protocol. Pluto may provide additional support to make file system operations recoverable; this idea is currently being investigated.

Bluto Server. The Pink Personal AppleShare service [5] will implement one example of a Remote File System Agent, a software component that offers file service from the local machine to a network. Remote File System Agents rely on Pluto for local file service, which they then export to their network.

High Degree of Reliability

The file systems available under Pluto will implement their on-disk structures so that they are recoverable; that is, so that they can be reverted to an earlier consistent state in the event of an unexpected failure. There will be no need for a separate scavenging utility, like Disk First Aid.

High Level of Performance

Pluto provides various classes of service intended to deliver performance to different kinds of clients. Control over file system policies, like those for disk space allocation and data buffering, is made available through the client interface, for those clients that reject the default policies.

Heterogeneous File Systems

The greatest functional requirement of the Pink system is the need to support a mix of heterogeneous file systems. Media written by other file systems, and network file service offered by other systems, can be imported into the Pink System through the Pluto guest file system facility. The heterogeneity of guest file systems is masked for basic file system operations, so that all file systems appear equal when it comes to storing documents and applications.

Overview

The architecture of Pluto is based on the client-server model. A *client* is a program, an Opus/2 team, acting on the behalf of some human *user* of the computer, and a *server* is a team providing some service to a set of clients. Discourse between the client and the server is accomplished through interprocess communication. The software that implements clients and servers in Pluto is based on the SCREAM [6] client-server framework.

Pluto Interface

Clients express their requests for file service in terms of the *Pluto Interface.* The Pluto Interface is a set of classes for file system objects. Most objects with which clients deal are surrogates for real objects managed by a local server known as a *File System Server.* Operations performed by clients on their surrogate objects are transformed into requests to the server owning the real object. To improve performance, the system-level protocol between the client and the server is not implemented as a one-to-one mapping of the operations on surrogate objects. For example, buffering of information might be done on the client's side. Figure 1 shows the relationship between an application program and the Pluto Interface; Figure 2 illustrates the relationship between client and server.

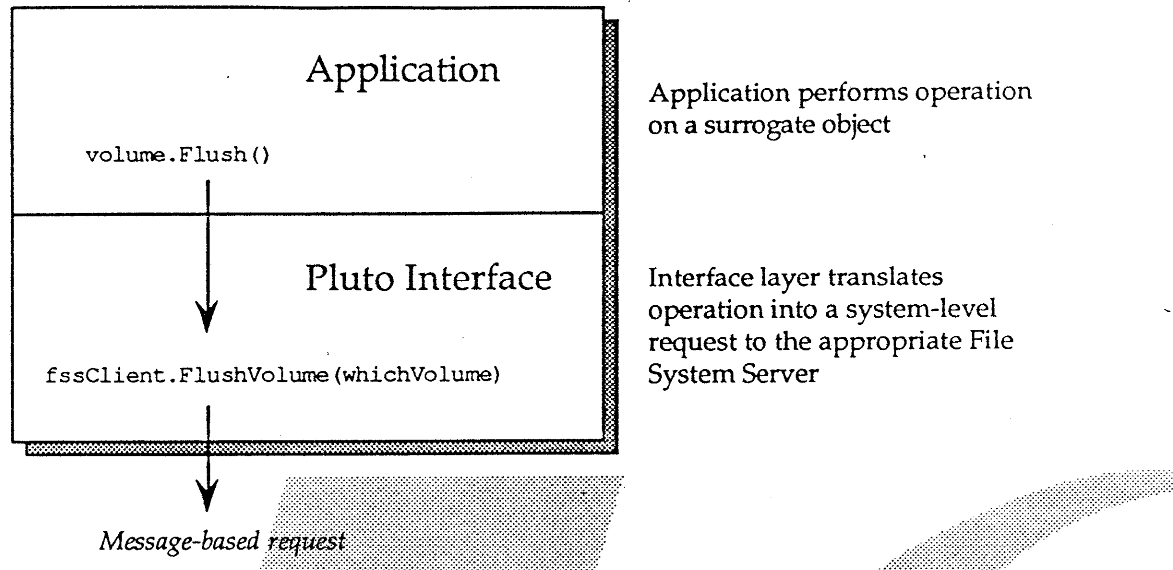


Figure 1. Relationship Between Application and Pluto Interface

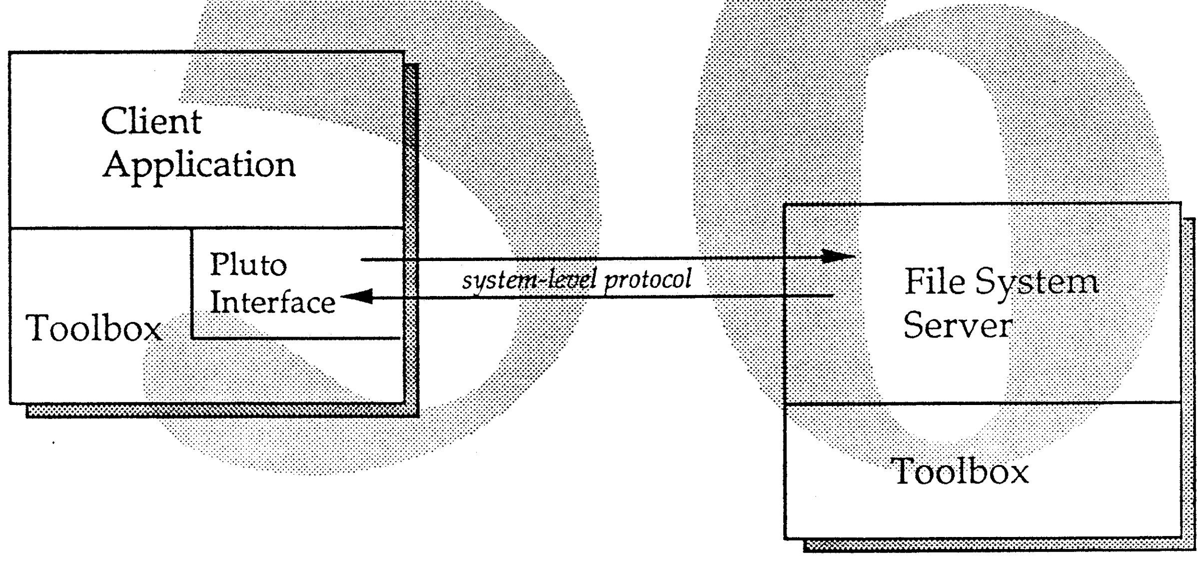


Figure 2. Relationship Between Client and File System Server

The Pluto Interface is designed for the file systems of the next decade; all file sizes and offsets are expressed in 64-bit quantities, allowing for a maximum size of 18 billion gigabytes, and all symbolic names are expressed using Unicode [7], the international 16-bit character set. The interface also makes regular use of the polymorphism of objects to build in extensibility. For example, lock types are expressed as objects, allowing for a new file system to introduce a new type of lock without necessitating a change in the interface.

The salient concepts supported in the Pluto Interface are file system objects (volumes, directories, file groups, files, and reference files); access control; logical names; user-defined properties; property queries; memory-mapped and byte-stream access methods; file-level and record-level locking; and user-defined file Content Servers.

Adapters

Opus/2 is a system kernel on which different operating system environments can be constructed. The software that implements a particular operating system environment is known as an *Adapter*. As far as the file system is concerned, the role of the Adapter is to "adapt" the Pluto Interface to the host interface of the operating system environment. Figure 3 depicts a foreign program obtaining file service through an Adapter.

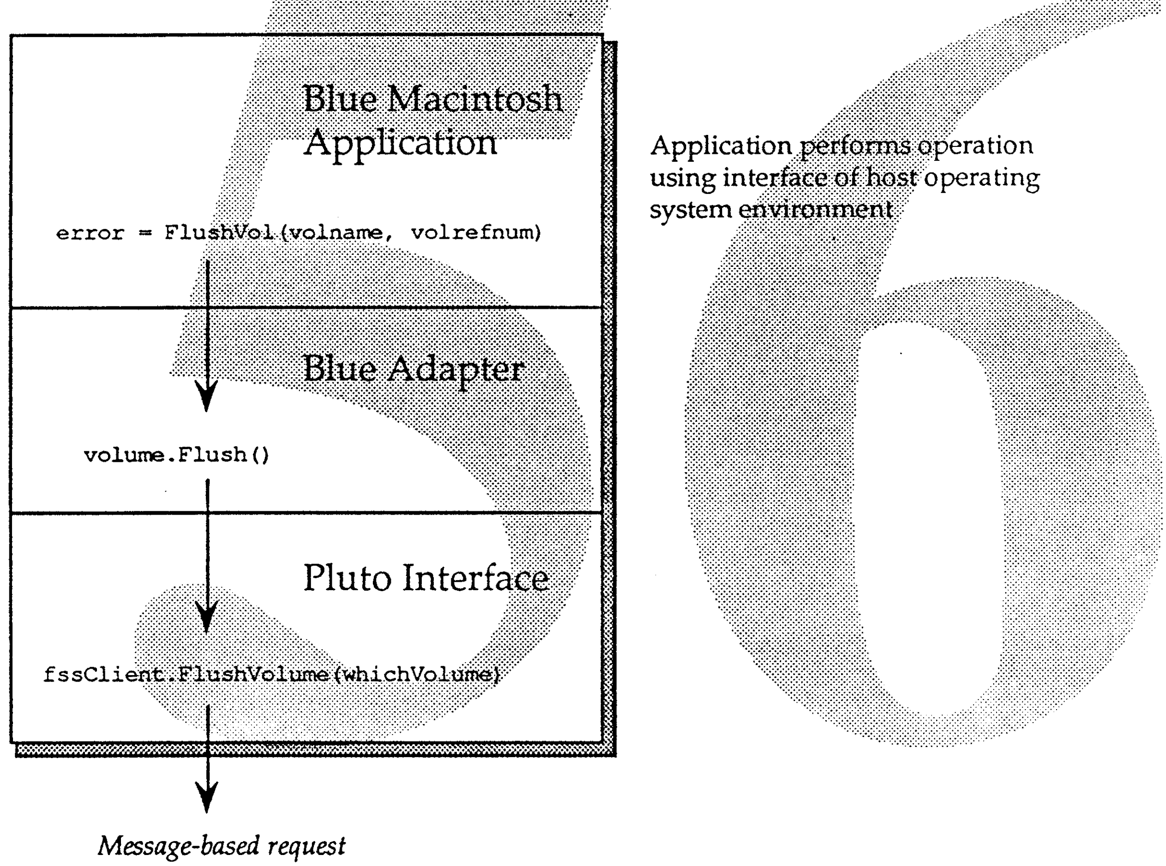


Figure 3. Relationship Between Application, Adapter, and Pluto Interface

File System Server

A File System Server is responsible for managing a set of volumes of a homogeneous format, for example, a set of HFS volumes or a set of High Sierra File System volumes. A File System Server is an implementation of a particular file system, and, hence, by having different servers available in the system, different file systems are supported simultaneously. This situation is depicted in Figure 4.

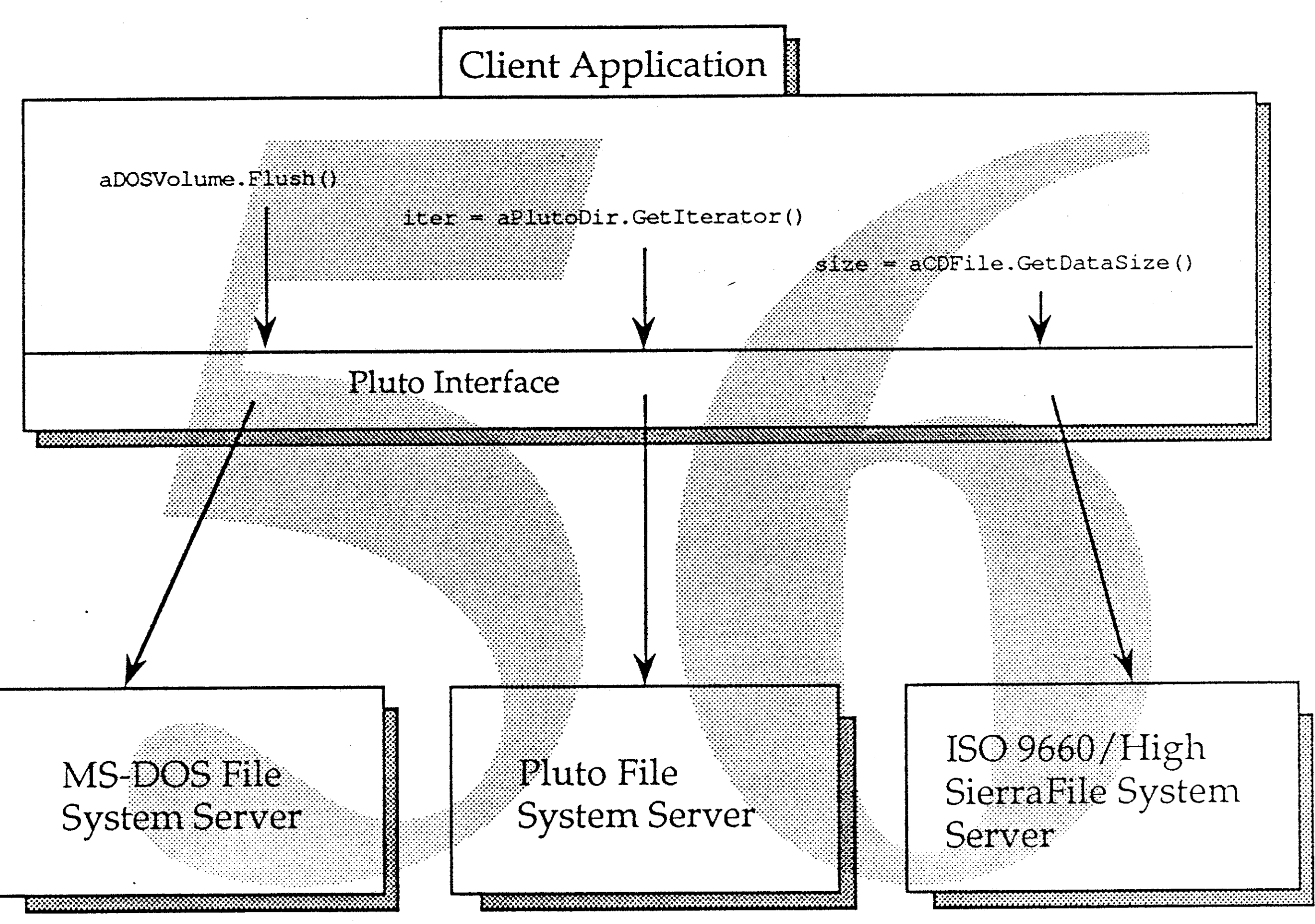


Figure 4. Application Using Multiple File Systems

Each File System Server is responsible for delivering a standard set of services. This set of services is defined by the Pluto Interface. In this light, each File System Server can be thought of as an adapter that adapts the services of the specific file system to the services described by the Pluto Interface. This scheme accommodates both local and remote file access. A File System Server for local volumes, that is, volumes on devices directly attached to the computer, will express each Pluto operation in terms of the on-disk structures of the specific file system, whereas a File System Server for a remote volume will express each Pluto operation in terms of the network protocol used to communicate with the network file server.

Any File System Server is free to implement second-tier operations in addition to the standard operations of the Pluto Interface. Second-tier operations are made available to client programs by introducing subclasses of existing classes in the Pluto Interface.

Some examples of local file systems are HFS, MFS, ProDOS, High Sierra, MS-DOS, OS/2 HPFS, and A/UX, while some examples of remote file systems are AppleShare, PC-NET (SMB), TOPS, Novell NetWare, and NFS.

Content Server

In addition to high-level file system services, every File System Server must also provide low-level data access services. These services are implemented by a component of the File System Server known as the default *Content Server.* The Content Server implements a set of file allocation and data transfer (page-in and page-out) operations used by the Virtual Memory Manager of the Opus/2 kernel.

On a file-by-file basis, the default Content Server of a File System Server can be replaced by a user-supplied Content Server team. A custom Content Server should provide "storage" semantics, so that two identical read operations with no intervening write will always return the same data, but is otherwise free to copy, translate, or manufacture the data that it provides to the kernel.

Pluto File System

A new on-disk structure, the *Pluto File System,* will be designed that will directly record the extended information necessary to support the Pluto Interface (e.g., Unicode names, properties, and access control information), and that will support the storage of large and sparse files. Concomitantly, a new version of the Apple Filing Protocol (AFP) will be designed to directly support the Pluto Interface. These new file systems will be the native file systems for Pink machines, and, because they support the Pluto Interface directly, will afford the best performance and reliability of any file systems available for the Pink system.

Guest File Systems

File systems other than the native file systems are referred to as *guest* file systems. Guest file systems support volumes created by non-Pink host machines. Such volumes are referred to as guest volumes. The on-disk structures of a guest volume may not be capable of directly storing certain information necessary to support the Pluto Interface. When this happens, the information must be stored in a meta-structure, like a file. This file contains whatever additional file system information is necessary to make the guest file system look like a native Pluto file system, and is referred to as the *Pluto Database.*

The discussion so far has related to the on-disk data structures of a local file system, but the same line of reasoning can be followed with respect to remote file systems. When there is no direct support in a remote file system's protocol for the storage of a particular piece of information, that information must be stored in a Pluto Database on the remote volume.

Since the Pluto Database is accessed using client-level mechanisms, the integrity of the database is difficult to guarantee. Apart from the obvious data-loss problem should the database file be overwritten or expunged, there is also the problem of consistency to consider; whenever a guest volume is shared with non-Pink systems, the information stored in the Pluto Database can become inconsistent with the actual state of the volume.

Various implementation techniques can be brought to bear to improve reliability, like on-the-fly consistency checking and replication of the information in the database, but they all impose a penalty on the overall performance of the file system. Each guest file system implements a file model for the benefit of its clients, and the logical distance from that file model to the Pluto model determines the cost of translation.

Remote file systems pay an additional penalty when high-level operations, like a directory search based on a property query, cannot be executed at the network file server because the filing protocol cannot express the operation. In such cases, rather than migrating the computation to the network server, the data involved in the computation must be migrated to the local machine.

File System Server Framework

A *framework* is a "set of classes that embodies an abstract design for solutions to a family of related problems, and supports [software] reuse at a larger granularity than classes." [8] Since much commonality is expected across implementations of File System Servers, it is hoped that the sharing of code and implementation effort can be promoted by developing a framework for the construction of such servers. The code of a particular File System Server, then, would consist of the server framework coupled with a file-system specific part.

The goal of the File System Server Framework is to reduce the cost of integrating new file systems into the local environment.

Remote File Access

Customers can integrate a Pink machine into their pre-existing distributed and network file systems once a File System Server has been developed for their file system.

Pluto supports an import-export, or remote-mount, model of remote file access. Volumes can be imported individually from remote file systems that export a collection of directory trees, like TOPS and RFS. Distributed file systems, like Sprite, that implement a network-global directory tree can be imported as a single volume.

Directory trees can be exported from the local machine to a network with the help of a *Remote File System Agent* for that network. The Agent represents remote clients who wish to access file system resources on the local machine. Remote File System Agents are clients of Pluto, and are not, therefore, a part of the Pluto architecture; however, they rely heavily on Pluto services.

The import-export model allows the user to explicitly control the nature and the extent of file sharing, and to control the consumption of local resources.

Volumes on the Desktop

From the user's perspective, the desktop is populated with objects that can be opened through direct manipulation. One object might be a local volume, which can be opened to reveal the file system objects in the root directory. Another object might be an AppleTalk network or an NFS network, either of which could be opened to reveal the next semantic level (zones in the case of AppleTalk and domains in the case of NFS). Authentication might be required when a protected object is opened, for example, opening an AppleShare server would require authentication before the volumes within it could be viewed. Whether the user has to type a password at that moment is determined by the policy for caching authentication information locally.

Local volumes are discovered at the time the system starts up. As device Access Managers begin executing, they register each logical device with the Desktop Manager. The Desktop Manager, or the Access Manager, then calls Pluto (TFileSystem) to mount the volume, specifying the logical device. Mounting the volume establishes a connection so that the data recorded on the volume can be accessed. The mount call returns a volume object (TVolume).

Pluto will cycle through the available File System Servers to find one that recognizes the file system image written on the logical device. If no File System Server responds favorably, an exception is raised. A similar sequence is performed when an Access Manager detects that new media has been inserted into a removable-media drive. Note that at boot-time there is a circularity; an Access Manager and a File System Server must be in place in order to launch a File System Server, unless one or more distinguished File System Servers is recorded in a ROM or in a boot image.

Remote volumes are discovered by browsing the networks attached to the computer (by opening desktop objects representing zones and file servers). Control is passed to a File System Server at the point that a volume to be mounted has been located and identified. Authentication to the network file server holding the volume occurs outside of Pluto. If communication with the network file server requires the establishment of a virtual circuit, then that is also done outside of Pluto. A communication object for interacting with the network file server is passed to Pluto, along with an identifier for the file system type.

A volume mount operation may fail because of access control restrictions associated with the volume itself; for example, if the volume is protected by a password. Such a failure will raise an exception to be handled by the software trying the mount operation.

File System Objects

Surrogate Objects

Most of the file system objects defined by the Pluto Interface classes are surrogate objects. A file object held by a client, for example, is a surrogate for the real file stored on disk. A surrogate object is a description of the real object, used to identify the real object to other system functions. Holding a surrogate object is the equivalent of holding a name. When the real object is deleted, the surrogate object will be invalidated, and operations performed against it will fail.

Volumes

A volume object (TVolume) represents a mounted volume. Volume objects are acquired by interacting with a seminal object (TFileSystem) that represents the total file system (i.e., the aggregate of File System Servers). The seminal object can return volume objects in three ways. First, in exchange for a mount descriptor object (TMountDescriptor). A mount descriptor encapsulates the file-system-dependent information necessary to establish a connection to a volume. The mount descriptor might contain information like a password object, an object representing the storage provider (e.g., local device or network session), or a file system type object (e.g., NFS or AppleShare). Second, in exchange for a volume name. Third, indirectly through a volume iterator, which can be used to obtain a surrogate object for each mounted volume.

A volume object can be asked to return a surrogate object for its root directory. Starting at the root directory, clients can navigate through the hierarchy of directories on the volume.

Local Volumes

A local volume is recorded on some block-storage device attached to the local machine. The file system volume seen by clients is an abstraction implemented by a File System Server; the File System Server, in turn, deals with a *logical volume* abstraction implemented by a device Access Manager.

A logical volume is basically an array of blocks; though, the Access Manager permits byte-level addressing of the data on the logical volume. A logical volume may correspond to a partition on a physical volume, or to multiple physical volumes (i.e., an array of disks). The logical volume may be a stable volume, implemented as a mirrored physical volume. The Access Manager hides the exact implementation of the logical volume. The Access Manager can, however, return a geometry object that describes the significant parameters of the underlying physical volumes.

The Pluto File System will be designed to allow for file system volumes that comprise multiple logical volumes (a volume set). This capability would enable the storage of a file larger than a logical volume and the incremental addition of storage to an existing file system volume. The feature will not be implemented for the first release of Pluto, though.

Directories

Every volume stores a hierarchy of directories. File system objects stored within the hierarchy of directories can be discovered by searching from a given directory; the directory object is asked to look up the object based on a symbolic path name relative to itself. If the object is located, a surrogate for it is returned. A description of symbolic path name objects can be found in the section on Naming.

A directory object can supply an iterator that will enumerate the file system objects contained in the directory. Using a property expression as a filter, the iteration can be set up so that only objects of interest are returned (see the discussion of Properties).

File Groups

A file group is a directory subtree rooted at a distinguished directory known as the *parent* of the file group. A file group affords a tighter bundling of file system objects than can be achieved using the normal directory mechanism, and is intended to represent a single entity to the file system client. For example, the access control list of the file group parent overrides the access control lists associated with any of the file system objects in the group.

File groups can be nested; the interior file groups are considered to be part of the outermost file group until they are extracted. Any member of a file group can return a surrogate object for the parent of the file group. The file group parent can be asked to return information about the file group itself, like its aggregate size. Operations like these will be very fast in the Pluto File System, but will be more expensive on guest file systems (e.g., computing the size of a file group might require traversing the group).

File groups will be used in the same way that twin-fork files are used in the Macintosh today, except that instead of two elements, there is an unbounded number, and each element can be any type of file system object subordinate to a volume; for example, a member of a file group can be a reference file (symbolic link).

Files

The file system object that provides data storage is a file. A distinction is made between an open file, one whose contents is being accessed, and a closed file, one that is merely resident on secondary storage. An open file is represented by a *file handle* object (TFileHandle), which is created from a surrogate file object. A file handle enables operations on the contents of a file, like data transfer, space allocation, and byte-range locking. A file handle represents a connection to the file, so that deletion of the file is inhibited. Destruction of a file handle causes the associated file to become closed, unless there are other outstanding file handles referring to the same file (also, if the file has been memory-mapped, the close is deferred until the segment is taken down).

The success of an *open* operation is predicated on the acquisition of a particular type of file-level lock. File-level locks control the type of sharing that can occur while the file is open. Figure 5 shows the kinds of file-level locks that are supported. The access permissions (read or write) requested at open time are encoded in the file handle object; so the file handle is a capability for access to the file. For local file systems, changes in the access control information associated with a file will not revoke existing file handles. Remote File System Agents may want to recheck permissions on every access; so this behavior may be provided as an attribute of a handle.

Shared	*multiple readers and multiple writers*	read-write; deny none
Exclusive Read	*single reader*	read; deny read-write
Exclusive Write	*single writer*	read-write; deny read-write
Protected Read	*multiple readers or one writer*	read; deny write
Protected Write	*multiple readers and one writer*	read-write; deny write

Figure 5. File-Level Locks

A file handle is associated with the team that creates it by the File System Server. If a team dies before destroying its file handle objects, the death-watch recovery code of the File System Server will close the files. Because file handles are owned by a team, the tasks on a team can share file handles.

Another type of file handle, a *global file handle,* can be shared between teams. A global file handle exists independent of any particular team, and is recovered by the managing File System Server only when a team explicitly closes the handle. For this reason, care should be taken that a global file handle object does not become garbage. A global file handle enables cooperating teams to finely control the way in which a file is shared; for example, the Credence software could use this mechanism to deny access to non-transactional teams. A global file handle cannot be passed across a machine boundary. Whether global file handles should be counting is a question currently being examined.

Temporary Files

A temporary file is used to store data that is volatile across system restarts. A temporary file is anonymous and can only be referenced through a surrogate object. The file system is not used as a junk heap for name resolution. A temporary file object can be passed from one program to another (e.g., from compiler to linker). The secondary storage associated with temporary files is reclaimed when the system restarts.

Various performance optimizations are made for temporary files. Since temporary files are not permanent, storage allocation and data write-back are performed in a lazy manner, in the hopes that the file will be destroyed before the actual work must take place.* In addition, when temporary files are allocated in a sparse manner, the file allocation on disk is actually done sequentially, relying on the virtual memory page tables to preserve the correct order of the file pages. This strategy greatly reduces the cost of sparse temporary files on guest volumes that do not support sparse files. Sparse temporary files will become more common once the Opus/2 kernel begins to make use of copy-on-write segments.

* A close-and-delete operation allows the same optimization to be made for permanent files.

Immutable Files

A file can be made immutable; once this is done, the file can never again be written. An immutable file can be read (copied) or deleted.

Reference Files

A reference file is a file system object that is actually a reference to another file system object. Reference files are used by the Pluto File System to implement symbolic links. Reference files, as a general mechanism, can be used to implement other kinds of references, like auto-mount points that are references to remote volumes, or "caching" references to remote files that, when opened, copy a version of the remote file onto a local disk.

Reference files introduce indirection and special processing to the regular name processing done by a File System Server. Implementation of the kinds of reference files mentioned above is up to individual File System Servers.

Naming

Symbolic Names

The name space of file system objects is hierarchical, with the topmost level being a flat collection of volumes. The name space can be thought of as a forest of directory trees. As mentioned earlier, volumes become available by being located and then mounted; at a given time, a particular set of volumes is accessible. Networks are viewed as a source for remote volumes that can be mounted. This model is distinctly different from that of networks as the backbone of a distributed time-sharing system; for example, user mobility, the ability for any user to sit down at any computer and access his files, is not a goal of the file sharing model.

Path Names

A symbolic *path name* describes a path through the hierarchy of directories on a given volume, starting at a particular directory. Path names are always interpreted relative to an existing volume and directory; volume names do not appear in path names. Path names are represented as an array of textual strings. Adapters are responsible for parsing host path names into this canonical form. A distinguished entry is used in the canonical path name to indicate the immediate parent directory.

Adapters can participate in path name interpretation by performing separate directory look-up operations on each component of a path name, rather than passing the entire path name to the File System Server for interpretation. The UNIX Adapter would do this for programs that have set their root directories to something other than the global root directory.

Path name interpretation is typically performed once in order to acquire a surrogate object, and then further operations are expressed in terms of the surrogate object. Since path name interpretation is not performed on every operation, verification of the access permissions on ancestor directories is also not performed. This is usually acceptable, but if a local file system is being exported by a Remote File System Agent, such bypassing of directory-level access control might not be tolerable. Whether path access permissions are checked for every operation on a surrogate file system object should be an attribute of the object.

Pluto does not support the notion of a working directory. Clients can acquire and hold surrogate objects for any directories of interest.

Add Names

Additional names can be assigned as aliases for a file system object in a particular directory. The aliases are managed as properties of the file system object, except that all aliases within a directory are taken from the same name space. The Bluto Server will use this facility to store minimal names for file system objects being exported to MS-DOS systems.

Logical Names

The low-level identifiers provided by most file systems name the objects, or containers, implemented by the file system. This is often not the right abstraction for clients of the file system, who are more interested in naming the contents of an object, and not the container itself. For example, low-level identifiers of this sort cannot be guaranteed across backup-and-restore operations because the restored files are potentially placed in new containers, and therefore have new low-level identifiers.

To address this problem, Pluto provides *logical names*. A logical name is a non-symbolic name for a file system object. The logical name of a file system object is unaffected by changes in the object's symbolic name. A logical name is a property of a file system object, and therefore can be assigned by clients.

A logical name is constructed from a tag part and a hint part. The tag part consists of a time stamp and a random component for sparseness, such that there is virtually no chance that the name will be reproduced during the lifetime of the system. The hint part is used to improve performance, but is not necessary for the correct operation of the name; it will generally contain the low-level identifier supplied by the host file system implementing the named object.

Logical names will be cheap in the Pluto File System, but supporting them in guest file systems will be more expensive. For this reason, every file system object is not created with a logical name. A logical name for an object must be constructed explicitly. A file system object can have at most one logical name at any given time.

File Logical Names

The logical name of a file or directory can be used to uniquely identify that object in the context of a volume. A globally unique identifier for a file or directory can be constructed by combining its logical name with the logical name of the volume on which it is stored. Note that such an identifier does not support migration of the file or directory away from its home volume.

Volume Logical Names

The logical name of a volume encapsulates a file system identifier that selects the File System Server for the volume. A volume can be found based on its logical name; Bluto will extend this search capability over the network.

Properties

A property is a named, typed value associated with a file system object. The value of a property can have a built-in type or a class type. All file system objects have member functions (MFileSystemProperties) to retrieve and set the values of properties associated with them. Properties are named using textual strings. Since all properties are typed, an exception is raised whenever there is a type conflict that cannot be resolved with a conversion. A set of properties can be named in a list (TPropertyPointerList) that can be used to retrieve or set those properties in one operation; this is particularly efficient when used in conjunction with a directory iterator.

Property Expressions

A property expression (TPropertyExpression) can be constructed and used as a filter to select file system objects that satisfy the expression. The comparative operators (<, >, <=, >=, ==, and !=) can be used to construct simple predicates, like the following:

```
TFileProperty("file size") > 1024
```

Such simple predicates can be strung together using the logical operators `||` and `&&` to form expressions like the following:

```
TFileProperty("file size") > 1024 && TFileProperty("file size") < 2048
```

The object that represents a property expression is built up using operator overloading.

A property expression can be used to restrict the file system objects seen when searching a directory or directory subtree. By using a property expression to guide the search, the computation can be migrated to the File System Server (and, indeed, perhaps to the network file server). The number of context switches between client and server is then minimized.

Note that some queries will be very expensive to process; however, if a query can be optimized, the File System Server has the best advantage.

System-Defined Properties

System-defined properties are those properties of file system objects known to the managing File System Server. These properties are generally stored in the on-disk structure of the file system. There is a mandatory set of system-defined properties that all File System Servers must support.

When a file system object is copied across heterogeneous volumes, it is often desirable to preserve the system-defined properties of the originating file system. Pluto supports this by maintaining a per-volume lightweight database of properties. The size of an individual property will be limited (less than two kilobytes), allowing for optimizations in the access method implementation.

The system-defined property database is also used to make existing on-disk structures extensible, since new attributes can be added at any time.

User-Defined Properties

Utilizing the same mechanism, Pluto allows clients of the file system to assign new properties to file system objects. These user-defined properties can be used to hold any kind of information about the object; however, if a large piece of information is to be stored in association with a file, the file group mechanism should be used.

If a user-defined property will be quoted often in property expressions on a particular volume, then the volume can be advised to attempt to accelerate those queries (by maintaining a secondary index on the property of interest). The cost of updating the secondary indexes is paid by the applications manipulating those user-defined properties. Secondary indexes will not be constructed for system-defined properties.

As of this writing, there is growing support for a system-level database associated with the Desktop Manager. Such a database might obviate the need for a complete user-defined property mechanism in the underlying file system.

Protection

Policy

The protection policy for file system objects on a volume is set by the File System Server managing the volume. For guest volumes mounted locally, the File System Server will generally honor the foreign protection information recorded on the volume.*

File Service Domain

When discussing protection across heterogeneous file systems, it is convenient to introduce the concept of a File Service Domain, an autonomous administrative environment with its own implementation of file service, name service (for exported resources), and authentication service. An AppleShare File Server is a good example of a File Service Domain, as it implements its own authentication database (containing user, group, and password information) and naming database (containing information about exported directories). Another example would be an NFS network in which a Yellow Pages name service provides a common authentication database and naming database.

---

* Similar behavior is anticipated when A/UX volumes are mounted under the Blue Macintosh operating system.

Authentication

In order to access resources in a particular File Service Domain, a client must be authenticated in that domain. Authentication is the process of ensuring that a client is authentic, and should therefore be granted the access rights associated with its identity. Authentication is based on the client's ability to produce information proving its identity, and on the server's ability to determine that the information presented has not been forged. The information supplied by a client is referred to as its credentials. Credentials are ultimately tied to some human user who has, at some point, demonstrated his own identity, usually by tendering a password (though other mechanisms can be employed, like magnetic card keys). Credentials contain, at a minimum, a unique identifier for the user within the domain in question.

The user will identify himself to the File Service Domain containing the resources in which he is interested. If the File Service Domain is implemented by a single server, then the unique user name and password are passed directly to the server. If the File Service Domain contains many servers, then the user name and password are passed to an Authentication Server for the File Service Domain, and a set of credentials is returned. In NFS Release 2.0, for example, the Authentication Server executes on the local machine, reads a network-shared authentication database, and grants the client process a set of credentials that contain the unique identifier of the user. These credentials can be passed to any server in the domain. RFS implements a similar approach, except that a local authentication database is used, and local credentials are mapped into new credentials when they are passed from client to server across a machine boundary. In either system, passing the credentials in the clear over a network reduces the overall level of security of the system to that of the network.

Regardless of the scope of the credentials, they may have various properties associated with them, like an expiration time, a time at which they become valid, and a list of the group affiliations of the user.

The next two sections review the topic of authentication as background for upcoming discussions; the reader already familiar with authentication techniques can skip these sections.

Network Authentication

When credentials will be transmitted over an insecure network, the Authentication Server can return credentials that are good only for presentation to a particular server. This approach involves protecting the credentials, generally by encrypting them with a key known only to the Authentication Server and the server to be contacted (the server's private key). In this way, the server presented with a set of credentials can determine if they were issued by the trusted Authentication Server. When a client requests credentials for a particular server, the reply from the Authentication Server is encrypted using the client's private key; so only the client can extract the credentials from the reply. The client's private key is a reduced form of the password supplied by the user. The reply contains another encryption key, the session key, which can be used for secure communication once the session between the client and server has been established. The session key is replicated in the credentials, so that it can be safely passed along to the server. To prevent replay of captured credentials, and to authenticate the client, the server can issue challenges that require the client to demonstrate knowledge of the shared session key.

The scheme just described relies on conventional encryption, using a single private key. A similar level of protection can be provided by schemes that use public-key encryption, where public keys are used to encrypt messages sent to clients or servers that can decipher them using a private key.

Local Authentication

When credentials are to be communicated only within the bounds of a single machine, they can be safeguarded by the operating system kernel, relying on the hardware protection boundary implemented by the user-supervisor trap mechanism. This obviously works when the file system is implemented as a part of the kernel. When the file system is implemented outside of the kernel, the IPC service provided by the kernel must be made secure. This can be done by having the kernel stamp messages with an non-forgeable identifier, from which the user on whose behalf the client is executing can be deduced, or by using a scheme like that described for the case of network authentication. In either case, a trusted third party is involved, at least to collect the user's password and to grant credentials (user identification) to client teams executing on behalf of the user. This third party is the *Local Authentication Service,* a part of the Pink system that has not yet been fully defined. For now it is sufficient to assume that the identity of a client sending a message to a File System Server can be reliably determined.

Partitioned Authentication

Before turning to a discussion of access to remote File Service Domains, there is one aspect of local authentication that warrants further analysis. Volumes attached directly to the personal computer do not implement their own File Service Domain, yet they may rely on membership in a File Service Domain for their correct operation (i.e., they don't store their own user-group database, but they store protected files that refer to one). The interesting question is whether, or how, to honor protection information on a volume when the volume is partitioned from its File Service Domain. For example, this situation can arise when the File Service Domain relies on the network authentication service, and the network authentication service is unavailable. When this happens, the alternatives are to authenticate all clients as guests (with "world" rights), or to authenticate clients based on a locally cached subset of the authentication database (based on the assumption that availability is more important than absolute security, since the cached information can be stale). For volumes to be movable from one machine to another, the latter approach would require that the cached authentication information be stored on a per-volume basis. For example, a volume could be transported from one File Service Domain to another, disjoint domain in which the user moving the volume is unknown and cannot be authenticated. If normal, protected access is to be enforced, the volume itself must contain the necessary authentication information. In this case, the volume implements its own degenerate File Service Domain.

The area of authentication, and the Local Authentication Service in particular, is one where more design work is necessary for the Pink system. Human interface concerns will, no doubt, guide this work. The Pluto architecture provides mechanisms that are general enough to accommodate whatever system design is adopted.

Authentication for Remote File Access

Remote file access requires authentication between the File System Server, as an agent of the client, and a Network File Server. Remember that, for connection-based network file servers, a preestablished session is handed off to the File System Server. If this session is not multiplexed across multiple clients, then credentials are implicit in the session (this is the case for AFP). If the session is multiplexed, that is, it has been established from machine-to-machine rather than from client-to-server, then credentials must be associated with each client sharing the session (this is the case for RFS). If the network file server is datagram- or RPC-based, then credentials must accompany every rendezvous.

The credentials for clients accessing remote file systems should come from the Local Authentication Service. If multiple users can be represented by teams executing concurrently, then credentials must be grouped by user. A client team executing on the behalf of a particular user, and attempting to copy a file from one File Service Domain to another, must have valid credentials for each domain (though for some network file systems, lack of credentials will result in guest access). The local Authentication Server might also be able to map credentials from one File Service Domain into credentials for another.

Whether a File System Server retrieves credentials from the Local Authentication Service or gets them directly from the client is not central to the design. Credentials are opaque to Pluto; they come as they are from the authorization service of the particular File Service Domain, and might afford varying levels of security. For example, NFS Release 2.0 [9] relied on authentication at each client machine, and then passed the resulting credentials in the clear over the network. These credentials were vulnerable to all sorts of attacks, including fabrication by non-UNIX machines on the same network. A later release of NFS fixed these problems by introducing credentials protected by a public-key encryption scheme. A File System Server does not offer any stronger protection guarantees than those offered by the network file system it represents.

For communication with network file servers that establish a separate session for each client, multiple sessions must be handled by a single File System Server (if a user is to be allowed to simultaneously connect to the same server under different user names). The authenticated session is tied to a user identification, and represents the credentials.

Access Control

From the point of view of Pluto, the credentials do not so much identify the client as they identify the client's title to a particular set of access permissions. A *permission* is the right to perform a particular operation on a file system object. In Pluto, an individual permission is represented by an object. Permission objects can be collected together into a set of permissions (TPermissionSet). A set of permissions can be associated with a *principal,* which is an object that represents an accountable entity, like a user or a group of users. A list of (principal, permission set) tuples can describe the access allowed to a particular file system object, and is called an *access control list.*

Access Control List

The Pluto Interface permits an access control list to be associated with any file system object. Individual File System Servers may impose greater restrictions; for example, the Pluto File System does not permit access control lists on individual files.

A principal object encapsulates a user or group identification that can be tested against the user identification in the credentials of a client. For some file systems like NFS or Andrew [10], the credential object would contain a list of the groups to which the user belongs. For other file systems, the credentials would contain only the user identification, and the Authentication Server would need to be consulted for information about group memberships. In the former approach, changes in group membership are not apparent until the credentials are invalidated or expire. In the latter approach, since protected file operations may require an interaction with the Authentication Server, response-time bounds on those operations become hard to assess. The nature of credentials for the native Pluto file system, and for the Pink system in general, has not yet been determined. This aspect also has an effect on the human interface and the feel of the system.

Since a principal named in an access control list is an opaque object, it could be a key or a password as easily as it could be the identifier of a user or group. Any client quoting the password would be granted the associated permissions. Such an approach has been adopted as part of the FTAM file access standard [11]. Some file systems may take advantage of this flexibility.

An access control list is a general mechanism for achieving protection. Because the individual permissions are objects, opaque to the implementation, the Pluto access control list classes could be reused elsewhere in the Pink system to protect other types of resources.

Heterogeneity of Permissions

Every file system has its own particular, and often peculiar, set of access permissions for the objects that it implements. Dealing with this heterogeneity is a difficult problem. The local Pluto File System defines its own set of access permissions, discussed below, which are modeled closely after the access permissions defined by AppleShare.

A File System Server that wishes to be a good citizen should support the same access permissions as the local Pluto File System, with semantics as close as possible to the Pluto semantics. Additionally, the File System Server can choose to provide access to its own native permissions. The objects that represent access permissions (TPermission) are simply wrapped tokens, and hence have an underlying textual name by which they can be discriminated.

Pluto Permissions

The permissions defined by the Pluto File System are shown in Figure 6. Remember that the Pluto File System does not implement access control at the granularity of an individual file; so the file access permissions listed are simply a suggestion to implementors of File System Servers.

The Pluto permissions relate to the current AppleShare permissions as follows: the AppleShare see-files and see-folders permissions have been folded together (in practice, they are rarely set independently), a new permission has been introduced to control delete and rename operations (as suggested in the preliminary ideas published for AFP++ [12]), and a new permission has been added to control modification of the access control list itself.

This last permission has been included to allow a separation of access control mechanism, as reflected in the Pluto Interface, from access control policy. No notion of file or directory ownership is visible in the Pluto Interface. The mechanism for access control is a set of permissions associated with a principal by way of an access control list. One permission is permission to modify the access control information itself. A particular File System Server might implement the policy that only one principal, the owner of the protected file system object, can have this permission at any one time. Another File System Server might implement an ownerless policy, like that of the Andrew file system.

Likewise, no notion of groups or a primary group is visible in the Pluto Interface. The access control list mechanism allows for any number of principals (groups) to have access permissions to a particular file system object. Restricting the number of groups, or distinguishing a particular group, represents a policy.

When an ownership policy is in effect, each file or directory would likely have an *owner* property associated with it. Similarly, when a primary group policy is in effect, each file or directory would have an associated *primary group* property.

The access control policies implemented by File System Servers can vary with no change in the application's interface to the file system.

Directory Permissions	AppleShare Equivalent
Read Files	See Files + See Folders
Write Files	See Files + See Folders + Make Changes
Create Files	Make Changes
Delete (Rename) Files	
Modify Access Control List	

File Permissions
Read
Write
Delete (Rename)
Modify Access Control List

Figure 6. Pluto File System Permissions

Inhibitory Flags

Many file systems support inhibitory flags as a supplement to the regular access control mechanisms. These flags can be set or cleared by any client with permission to modify the properties of the protected object; so, for example, a setting a write-inhibit flag does not afford the same degree of protection as withholding general write permission. However, inhibitory flags are quite useful in preventing accidents, and in controlling access at a gross level. The native Pluto File System will support, at least, write, rename, and delete inhibitory flags on all file system objects.

Degree of Protection

Time-sharing systems and remote file servers have the advantage that physical access by those other than trusted administrators is prevented. Systems of this type can be made fairly secure. On the other hand, personal computers, where physical access to the machine is the rule, cannot be made entirely secure. Most would argue they should not be made secure, since the security checks would be a formidable impediment to getting any useful work done.

The security provided by the local Pluto File System, the measure of confidence that the integrity of the file system will be maintained, is good enough to prevent everyday programs from avoiding access control restrictions. An "everyday" program is one that does not set out to breach the security of the system, by installing a non-standard Interrupt Service Routine, for example, or by installing a File System Server that neglects to check credentials. Kernel-level debuggers, like OpusBug, can subvert the system too, and, obviously, no guarantees can be made once a volume is detached from the computer and transported to a non-Pink machine.

The Pluto File System implements an active protection mechanism, one that attempts to provide a security envelope that clients cannot penetrate. To thwart attacks of the nature just described, a passive protection mechanism, like file encryption, is much more effective.

Data Access

The Pluto Interface supports two methods for accessing the data of a file. First, any file can be mapped into the virtual memory of a team and its contents accessed through references to virtual memory addresses. Second, any file can be opened as a stream and its contents copied to or from the virtual address space of a team through read and write calls against the stream object.

No record- or text-oriented access methods are supported. These can easily be implemented through objects that are clients of the Pluto access methods. Similarly, device independence, or the ability to perform I/O without knowing the nature of the source or sink of the data (e.g., a file on disk, a serial line, or a network session) is implemented at a layer above the Pluto Interface. The Pluto access methods deal strictly with files on block-structured devices.

There are several ways a client can use the two basic access methods provided by Pluto.

First, a client can map a file into its virtual address space (TSegment) and access the segment directly.

Second, a client can use the stream operations of the segment object to copy data to and from the mapped file. This approach combines the streaming (copy) model of access with the inherent efficiency of memory-mapped files for handling small, unaligned data transfers.

Third, a client can use the stream operations of a Pluto file stream object (TFileStream) to copy data to and from a file that is not mapped into its own address space. In this case, the file may or may not be buffered by the Opus/2 Virtual Memory Manager, at the discretion of the client. Buffering is helpful for servicing small transfers from extremely large files, for which the overhead of memory-mapping is too great. To a real-time client attempting to move bulk data directly from a file into its own address space, buffering reduces the performance of the transfer.

Content Server

The Opus/2 Virtual Memory Manager administers real memory as a pool of page frames, most of which are backed by secondary storage allocated to permanent or temporary files. Consequently, all of physical memory acts as a cache over files that are open.

Every open file is associated with a Content Server that implements the basic data access operations required by the Opus/2 Virtual Memory Manager. These operations are the following:

- Read data from the file into real memory
- Write data to the file from real memory
- Allocate new backing storage to the file
- Return a File Map (map of the file's blocks on the logical volume)

In the general model, when the Virtual Memory Manager must move data between real memory and secondary storage, it sends a request to the Content Server. The request contains the following information: a descriptor for the real memory, a file identifier, an offset into the file, and a byte count. This interaction is depicted in Figure 7.

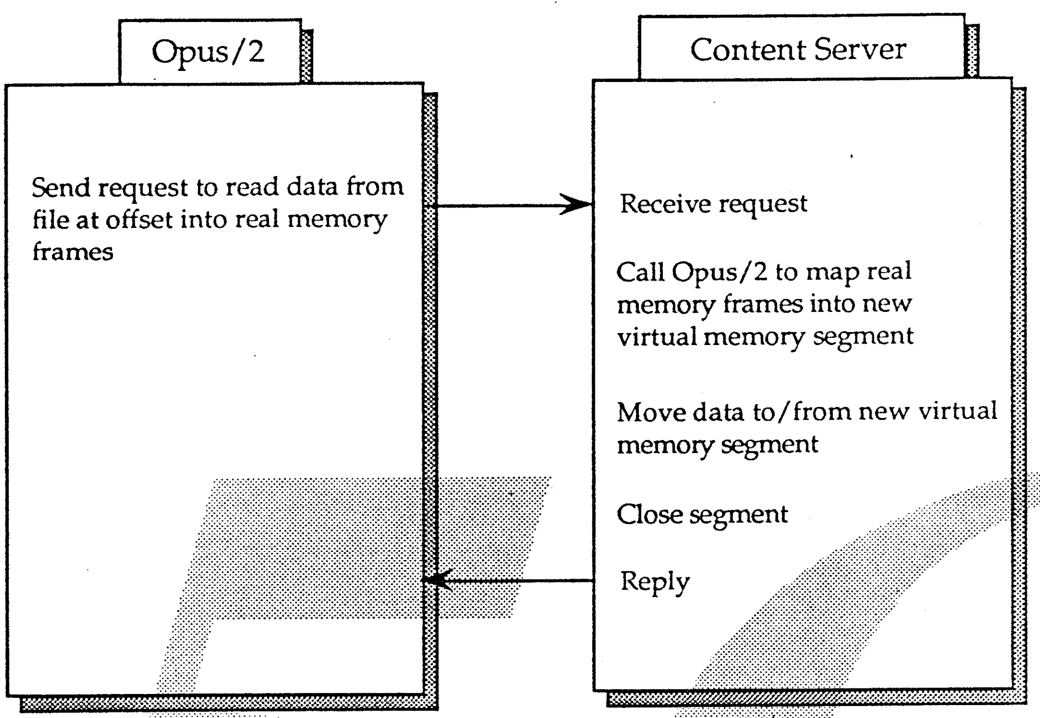


Figure 7. Opus/2 Request to File Content Server

Note that the Content Server acts as an intermediary between the Virtual Memory Manager and the manager of the storage medium. When the storage is implemented by a local Device Access Manager, and not a Remote File System Server or a custom Content Server, then optimal performance can be achieved by allowing the Virtual Memory Manager to cache the logical block map of the file in its page tables, and then go directly to the Access Manager when data must be moved. This arrangement is illustrated in Figure 8.

56

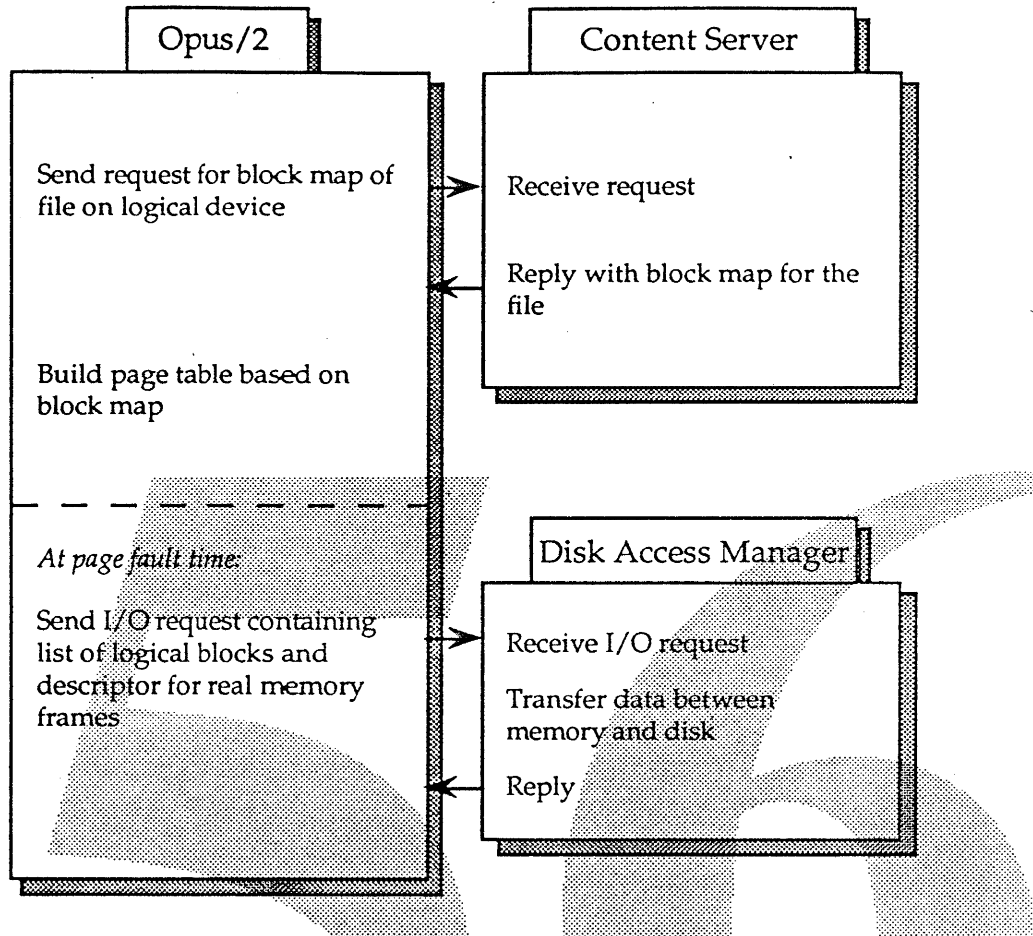


Figure 8. Fast Path for Local Page Fault

Custom Content Server

A client can indicate a custom Content Server when opening a file, and specify that the server will either manage the contents of the file or simply require notification when Content Server operations are taking place. The Credence software makes use of the notification service to implement its write-ahead log protocol. This relationship is shown in Figure 9.

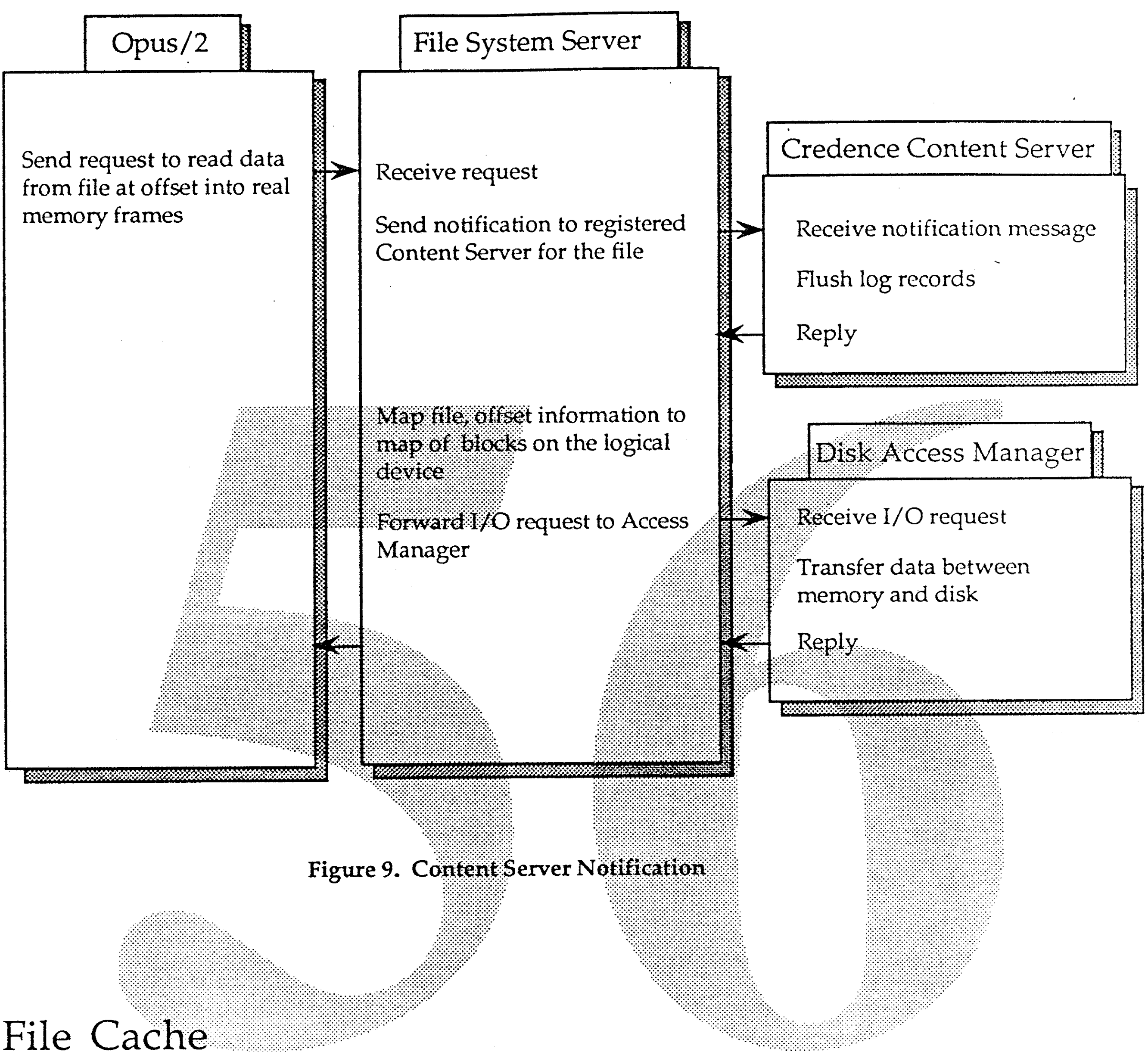


Figure 9. Content Server Notification

File Cache

Pluto does not implement a separate logical buffering mechanism for file data, but rather cooperates with the Opus/2 Virtual Memory Manager to manage the contents of real memory.

An open file may have caching enabled or disabled. When caching is disabled, the data of the file can be accessed only by using a Pluto stream object. Each read or write request goes to the Content Server for the file and the data is moved between the backing storage and the virtual memory of the client. When caching is enabled, the data of the file can be accessed through a Pluto stream object or by memory-mapping the file. When access is through a Pluto stream object, each read or write request generates a system call to the Opus/2 kernel.

The Opus/2 Virtual Memory Manager maintains a cache directory that describes what data from cached open files are present in real memory. The cache is filled by going to the Content Server for the file being accessed at the time the cache miss occurs. Cache misses can happen because of page faults (memory-mapped access) or because of read requests (stream access); the Content Server does not distinguish between these two types of access.

The Virtual Memory Manager, because it has dominion over the global cache, can institute policies that improve overall system performance, like deferring the copying of data to new frames as cache hits accrue to clients reading shared open files (i.e., a copy-on-write strategy).

Because there is a single shared cache for all file data, the consistency of shared file access is maintained for all clients executing on the local machine, regardless of what access method they are using.

Remote Data Access

Maintaining the consistency of shared file access for clients on different machines is the responsibility of each Remote File System Server. The File System Server must implement the cache consistency protocol of the particular remote file system that it serves. To this end, the Opus/2 kernel provides the following operations on the local file cache:

- Write back modified data of a file
- Invalidate all data of a file
- Write back modified data and invalidate all data of a file

Synchronization

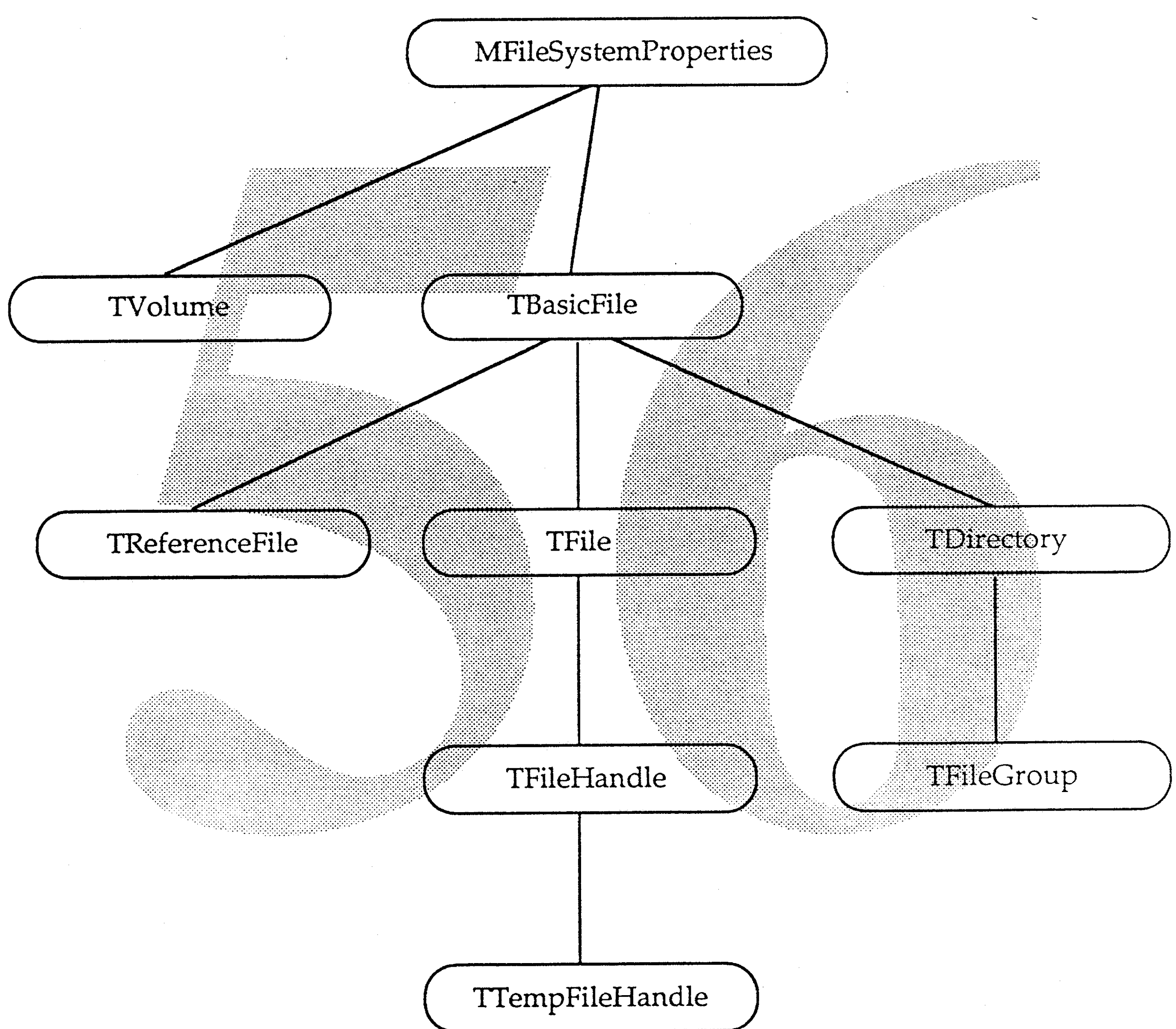
The Pluto Interface provides clients with byte-range read and write locks as a way for them to synchronize their accesses to concurrently shared files. The byte-range locks are only guaranteed to be advisory in nature. Range locks are non-blocking or blocking with an associated timeout set by the client.

Range locking is bound closely with cache consistency for remote file systems. Some cache consistency protocols require that all caching be disabled when there is more than one writer to a file. Memory-mapping, as an access method, can then only be supported in conjunction with range locking (modified data must be written back to the network file server when the lock is released, and the local cache invalidated). Pluto provides no support for application programs that require a consistent view of remote file data without explicit client-level synchronization.

Final Note

Anyone interested in having a copy of the complete Pluto Interface specification should request one from me. I can be reached by AppleLink at "MCFALL" and by QuickMail at "Crimson Permanent Assurance Co.:PINKTEAM:Chris McFall."

Class Hierarchy of File System Objects



References

[1] Bolton, L., D. Chernikoff, C. McFall, C. Moeller, and D. Orton. Opus/2: Memory, tasks, and IPC. Internal Document, Apple Computer, Inc., March 1990.

[2] Spiegel, J., et. al. Valhalla: Pink Finder. Internal Document, Apple Computer, Inc., March 1990.

[3] Spiegel, J., et. al. Odin: Desktop services. Internal Document, Apple Computer, Inc., March 1990.

[4] Neimeier, C. Credence: Concurrency control and recovery. Internal Document, Apple Computer, Inc., March 1990.

[5] Burns, G. Bluto: Pink Personal AppleShare. Internal Document, Apple Computer, Inc., March 1990.

[6] Orton, D. SCREAM: Servers, clients, requests, and messages. Internal Document, Apple Computer, Inc., March 1990.

[7] Collins, L. Unicode principles. Internal Document, Apple Computer, Inc., August 1989.

[8] Johnson, R. E., and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming* (June 1988): 22-35.

[9] Sun Microsystems, Inc. Networking on the Sun Workstation. Sun Microsystems, Inc., Mountain View, California, 800-1177-01, May 1985.

[10] Satyanarayanan, M., J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: Principles and design. *Proceedings of the 10th ACM Symposium on Operating System Principles*, Orcas Island, December 1985, 35-50.

[11] International Organization for Standardization. Information processing systems – Open systems interconnection – File transfer, access and management – Part 1: General introduction. ISO 8571-1, 1988.

[12] Venkatraman, R. C. AppleTalk Filing Protocol: Ideas and issues to support more file systems. Internal Document, Apple Computer, Inc., March 1989.

Bluto

Pink Personal AppleShare

56

The Pink AppleShare Team

Greg Burns x4-5465

1.0d5
March 15, 1990

56

Introduction

Bluto is Personal AppleShare for Pink. Personal AppleShare allows a Pink user to share files with other Pink and non-Pink systems. Bluto will support existing AFP servers and clients, and introduce support for Pink file system capabilities. This section presents an overview of Bluto, its features, and its place in the Pink file system architecture.

Goals

- Pink Personal AppleShare aims to provide transparent remote file system access. As such it is a fundamental building block of the Pink system, aiming to provide the user with the same experience no matter what file system is being used for storage, or its location on the network (within the limits of networking technology, i.e. performance).

- Pink Personal AppleShare aims to provide compatibility with existing AppleShare servers and clients -- Blue, third-party, and other.

- Pink Personal AppleShare aims to be scalable. The majority of the code used by Blue Personal AppleShare is shared by dedicated AppleShare servers. The core server implementation of Bluto will be shared between both personal and (future) dedicated servers.

- Market acceptance of Pink AppleShare is dependent upon availability of third party servers supporting its new features. Bluto aims to provide a portable code base for third party development of Pink-capable AppleShare services.

It is not a goal of Bluto to provide explicit support for server features such as stable storage. These features are deemed a part of the operating system, and are outside the scope of AppleShare services. Pink Personal AppleShare will support these features to the extent that they are provided by the underlying file system.

Architecture

Personal AppleShare consists of an AppleShare client and server. Both are Pink teams which will always be launched at system startup time. The client packages local file system requests into network AFP requests. The server handles network AFP requests, dispatching them to the Pink file system. The relationship between the Pink file system and Bluto is shown in figure 1.

Note that the Pink file system client could send its requests directly across the network to the Pink file system server, thus bypassing Bluto. We chose not to follow this design, as use of the Bluto client and server allows network performance optimizations to be made on the client side, and provides for compatibility with existing clients and servers; allowing remote access to heterogeneous systems.

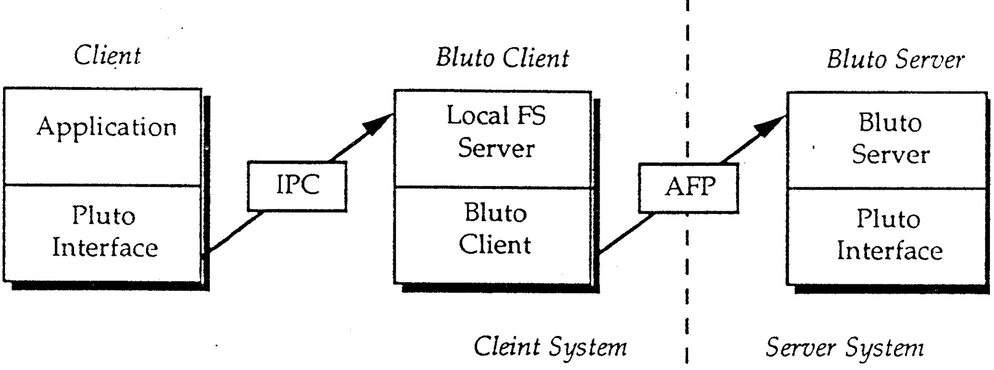


Figure 1. The Pink file system and Bluto.

The Client

The Bluto client can dispatch Pink file system requests across the network using AFP. Blue AFP versions 2.0 and 2.1 will be supported. In addition, a new AFP version, temporarily dubbed *k.0*, will be introduced. *k.0* will support the new capabilities supported by the Pink file system: properties, access control lists, etc. AFP *k.0* will, like the Pink file system, attempt to remain neutral in the features it supports.

We expect third-party servers to add support for AFP *k.0* some time after Pink AppleShare is released.

The initial version of the client will implement AFP 2.0 and 2.1. *k.0* will be defined as the Pink file system progresses. Since much of the file system mapping will be shared by HFS and AFP file system servers, this mapping will be implemented in a common framework, as a part of Pluto, the Pink file system.

The Server

The Bluto server will handle Pink file system requests from the network using AFP. Blue AFP versions 1.1, 2.0, and 2.1 will be implemented, thus supporting Blue 6.0, 7.0, and continuing to support the Apple II and AppleShare PC. In addition, AFP *k.0* will be introduced.

The server manages one or more local volumes. Volumes may be local disk volumes, CD-ROM volumes, or exported sub-directories. The Bluto server is a client of the Pink file system, making use of file system caching, access control, properties, etc. The ability of Bluto to serve HFS, High Sierra, etc. is dependent upon Pink support for these file systems.

It is a goal of the Bluto server to not maintain any parallel data structures.

Features

User Interface

Both the Bluto client and server require a variety of local services in order to implement Personal AppleShare. Among these are interaction with the Pink Finder, the Pink File system, the Pink Browser, the Pink Address Book, and the Pink Network Manager. Both Bluto client and server will also interface with a local authentication service for obtaining and verifying remote credentials. To a large degree, these Pink services will determine the user's experience of both the network and local file systems; the features that follow help to define the requirements of Bluto.

Bluto has very little human interface of its own. The interface through which the user perceives AppleShare is implemented by the Pink services mentioned above. Sharing, users & groups, and access rights will be visible through these services. Bluto provides their underlying support.

Startup

The Bluto server will start at system boot time, registering itself with the system name. It will always run in the background during normal system operation. Once server activity starts, the Bluto server will create a fixed pool of server tasks to handle network requests. The server will invoke the services of the file system,the authentication service, and the address book to complete these requests.

Since the server is always running, sharing a volume or folder will be a quick operation.

Shutdown

The server will never shutdown. To reduce system resource loads during periods of no server activity, tasks from the server pool will be killed after a specified idle time. A minimum number of server tasks (1 or 2) will always be running to accept incoming service requests.

The user will be able to set server parameters and explicitly enable/disable all sharing from a preferences or other such dialog. The latter will not shutdown the server, it will only disable sharing.

Volumes

The Pink file system is hierarchical, with the topmost level being a collection of volumes. Volumes may be physical volumes, such as a hard disk or CD-ROM, or network volumes. Bluto allows sharing of a directory sub-tree at any level in the hierarchy as a network volume. All or part of a volume can be shared.

The Pink file system also maintains file system privileges. The act of sharing a volume or sub-directory will be integrated with the finder interface. Sharing volumes and directories will be identical operations on both the local file system, Pluto, and remote file systems, Bluto.

The client will make a distinction between explicitly mounted volumes and volumes mounted via links. Auto-mounted volumes will be expired after a period of inactivity.

We would like to remove the limitations imposed by having AppleShare volumes appear as individual icons on the desktop. We expect links and authentication to result in an increase in the number of volumes mounted. One possible solution is to permit browsing of volumes and/or servers as an extension of directory browsing.

Accessing a volume through a file link (via desktop object or document link) should be completely transparent, with the authentication services performing the necessary user verification. Since many volumes may be mounted indirectly through links, we expect that such mounts will timeout and unmount after a period of inactivity. This should not result in any visual change to the user, and remounting the volume on subsequent activity should be transparent.

File Naming

AFP *k.0* will incorporate the Full/Minimal naming algorithm of AFP++ [Venk89]. Names are defined as attribute tuples of client file system names. The Full Name is the most permissive file name tuple. The Minimal Name is the most restrictive. Using this algorithm, we store only two names for each object in the file system, regardless of how many types of clients access the server (Four at this time - Pink, Blue, MS-DOS, and ProDOS).

For Bluto, the Full Name uses the Pink 16-bit character set. The tuples of AFP *k.0* use the following name attributes:

Client	Length	Character Set	Format Restrictions	Case-Sensitivity
Pink	255	Pink 16-Bit	None	No
Blue	32	Extended ASCII	None	No
ProDOS	15	Alpha-num	1st Alpha	No
MS-DOS	11	DOS	8.3	No

The Full Name is always the name assigned by the file creator. If a Blue client creates a file named 'Apples' on the server, 'Apples' will be the file's Full Name. The minimal name is derived by a server-specific algorithm, in this case it is also 'Apples'. If a Pink client creates a file 'Bread آبز' on the server, 'Bread آبز' will be the file's Full Name, and the minimal name might be '!Bread_?_.??'. The latter would be seen as '!Bread ? ??' to a Blue client.

There is no explicit support (or lack of support) for case-sensitivity. This attribute of file naming is server specific. Case-insensitivity is the preferred behavior, and the Bluto server will be case-insensitive.

Users & Authentication

Users and groups will be managed by the local address book and authentication service. This functionality has been separated from AppleShare, since these users, groups, and authentication will be used by services other than Bluto.

The address book should include a mechanism for accessing users and groups on ADAS, as well as allowing the user to create new users and groups which may not be in the ADAS database. The Bluto server will use these local users and groups in addition to the ADAS lists when checking access rights.

The local authentication service will be the system's point of contact with any network authentication services, Apple or otherwise. The user may have accounts on systems which do not use the authentication server. We would still like to have transparent mounting of these volumes, and expect the local authentication service to handle multiple authentication domains and storage of passwords.

The Bluto client and server will use the authentication service to establish authenticated sessions. This service should provide, or allow the incorporation of new authentication methods: e.g. Kerberos, Apple Random Number Exchange, etc.

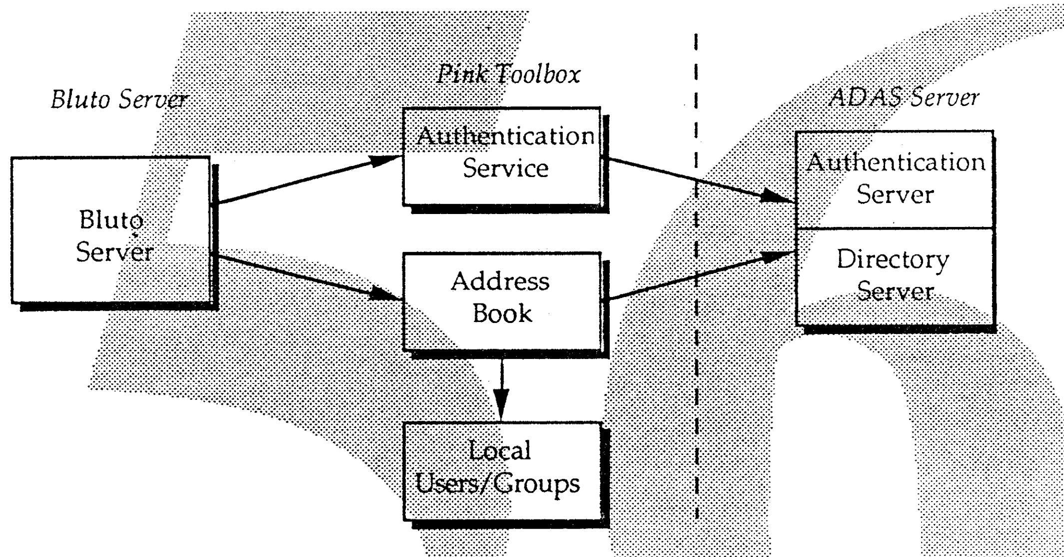


Figure 2. Bluto, Authentication, Users, and Groups

Authorization

The Pink file system, and thus Bluto, will support access control lists as the method of protecting file system objects. The advantage of access control list are their flexibility: directories can be shared with one or more users or groups. This lifts the constraints imposed by the protection bitmaps used in Blue.

Although the local file system will allow support for volumes with file level protection, Bluto will continue to provide protection only at the directory level. This 'protected container' paradigm is easier to work with, as users typically keep a mental map of where their files/folders are located, and their protections. Supporting file level protections would greatly increase the size of this mental map, making the system more difficult to use. AFP *k.0* will provide support for file level protection for servers implemented in foreign environments.

The semantics of users and groups are not defined by Bluto or the file system. The definition depends upon where the users and groups are defined – locally, in ADAS, or on a remote third-party system. Our goal is for the local and ADAS users and groups to have similar semantics (e.g. groups of groups, administration privileges, etc).

The rights for file objects are kept on the local file system, by Pluto.

Existing AFP 2.1 servers are easily presented by the access control list interface. Updating these privileges will be inconsistent for the user, since adding of additional shared users and groups is not permitted under AFP 2.1.

Object sharing and access rights should be well integrated with the Pink system. In particular, other objects which may have remote accessors (Remote Access, Collaboration, whatever.) should use the same permission model and interface. We must also handle the case where a foreign file system defines object protection capabilities not supported by the local file system or Bluto. There must be a consistent way to extend the access privileges user interface without sacrificing the consistency of the user interface.

Properties

Bluto will support properties as defined by the Pink File System. For AFP 2.1, these will be emulated. AFP *k.0* will include explicit support for file properties.

Caching

The performance and scalability of AppleShare is limited by the contention for the server CPU and disk[Lazo86,Saty85]. It order to increase the performance of AppleShare in large networks, we aim to reduce the load placed on AppleShare servers. For Personal AppleShare, this will reduce the CPU requirements the server processes place on the local machine. Reduction of server load will be done through intelligent caching of file data, properties, and names.

The protocol to be used by AppleShare is a slightly modified version of the Sprite cache consistency protocol [Nels88]. This algorithm guarantees a consistent view of data in the file system, even when the file is being cached by several workstations at once.

Files are distinguished as write-shared or non-write-shared. Non-write-shared files can be cached by the AppleShare client without danger of inconsistency. The server responds to each open request indicating whether a file is or is not write-shared. A single writer qualifies as a non-write shared file, and does not need to write-through to the server.

When a file is write-shared, no caching is done and all clients access file data directly from the server. Readers and writers are guaranteed the same level of consistency.

When a non-write-shared file is subsequently opened for write, the server initiates a callback sequence. Each client which has the file open is notified that the file is open for write, and all cached data must be flushed. Clients are notified sequentially to avoid dependency on specialized network features. Non-write-shared operations may be continued by outstanding clients until all clients have been notified. The open for write does not succeed until all non-write-shared clients have acknowledged the callback or timed-out.

There are no periodic cache-consistency probes. There is no explicit support in AppleShare for whole-file caching, due to its inefficiency on first use and on small I/O, and its requirement of a local disk.

Bluto will use a delayed-write policy, where blocks are written back to disk 30 seconds after being modified. This cuts write-back traffic by 20-30% [Nels88]. There will be no write-through on close. Internal version numbers will be used to verify consistency of data still in cache when a file is reopened. If the network connection is lost, the client is notified that the volume is no longer mounted, and its cache is flushed. The delayed-write flushing minimizes loss of data to blocks modified in the last 30 seconds.

This caching scheme requires a stateful server architecture such as AppleShare. AFP *k.0* will include support for write-sharing notification. On previous versions of AFP, only files opened with a Deny Write mode will be cached, since there is no explicit support for cache callbacks.

Note that caching cannot be disabled for memory-mapped files. A mapped file is, by definition, always cached. The only way to ensure consistency for mapped files is to lock a page before reading it. For mapped files, the Bluto client will flush a page from the client's cache immediately (atomically) after it is locked.

Directory and file id lookups may be cached independently of file data and attributes, in order to reduce server load. These lookup requests represent roughly 50-60% of all server activity [Srin89].

The local file system and the Bluto client will share the same cache.

File and Byte-Range Locking

Byte-range locking will include support for lock callbacks. Unshared files can lock ranges without accessing the server. The server will callback all of a client's locks when the file is opened by a second process. Lock callbacks are only supported by AFP *k.0*.
Locking timeouts are implemented by the file system. AFP *k.0* will also add support for lock timeouts. There is no deadlock detection in AFP. All locks are either immediately denied, or eventually timed-out, depending upon the lock request.

Recoverability

Bluto provides limited support for transaction log files to enable recovering from node crashes. Stable storage, in order to recover from media crashes, is not explicitly supported by Bluto. Bluto does not provide support for stable storage, other than that which is provided by the underlying operating system. Note that node crashes are far more common than media crashes.

Support for log files will be limited to explicitly disabling caching when opening a file. Caching will then be disabled for the client, and optionally by the server (server cache disabling is server implementation dependent). Bluto also provides support for forcing any file's cached blocks to be written to the server's disk. This is a synchronous flush, and can be used to guarantee that log files have been written to disk.

It should be noted that disabling client caching, and flushing of server cache blocks, will increase server load and degrade overall system performance.

Links

Volumes have a fixed identifier associated with them at creation time. Bluto will allow a volume identifier to be seen without server access.

AFP *k.0* will support the assignment and lookup of immutable volume and file names.

AFP Protocol:

Each AppleShare server encapsulates the file system objects on the remote system, and exports a set of operations on these objects. This set of operations is defined by the AFP network protocol.

For AFP *k.0*, the model of network encapsulation is being changed to parallel the classes and methods defined by the Pink file system. Classes remotely accessible are similar to the ones defined by Bluto. Instantiations of these classes may be located or created, accessed via class methods, and destroyed.

For those familiar with AMP, Apple's Network Management Protocol, the approach is very similar.

Eventually, with proper run-time support, both AFP and the server can be extended by referencing new classes, which are dynamically linked with the server, or new methods of existing classes. This is a great advantage in being able to extend existing products without new server releases, especially when many third-party implementations exist. The challenge is building this system to work in a heterogeneous environment.

In addition, the following features will be added in AFP *k.0* to support Pink features:

- 16-Bit server names.
- Volume and file IDs will be assignable.
- AFP naming will be changed to use the full and minimal naming algorithm of AFP++. The full names will be Pink 16-bit names on 255 character length.
- There is no explicit support (or lack of support) for case-sensitivity.
- AFP *k.0* will provide support for file properties. Well-known property names will be bound to numeric identifiers for performance.
- AFP *k.0* will replace directory bitmaps with access control lists. Files may have access control lists, depending on the server implementation.
- The protocol will transport independent, and should run over ASP, ADSP, ASDSP, or TCP.
- Cache callbacks for shared write files will be added.
- Lock callbacks for shared files will be added.
- File and volumes sizes will be increased to 8-byte words.
- Flat volumes will be dropped in AFP *k.0*.

Desktop Database

Bluto will implement the desktop database calls for compatibility with AFP 1.1, 2.0, and 2.1.

Network requirements

Bluto will continue to use ASP as its primary transport protocol. In addition, AppleShare will run over connection-oriented streams protocols such as ADSP or TCP. ADSP allows us to use larger block sizes than ASP, and has a better retransmission algorithm, but its retransmission of data uses more network bandwidth. Our ideal protocol is ASP with adaptive retransmission and larger block sizes. ASP gives us finer control over the transactions associated with one file, so that a file's response characteristics can be tuned by the client. This could allow us to adjust the protocol retransmission and error checking based on whether the file being read is error-intolerant (data), or delay-intolerant (sound, video).

High performance and a high burst bandwidth go without saying; but we'll mention them anyway.

Network block sizes will be 4K bytes, although we may experiment with larger sizes on LAN speed networks.

Summary

Bluto provides Personal AppleShare file sharing for Pink. It will be tightly integrated with Pink, and access to remote systems will be highly transparent. Much of the design is preliminary, and feedback is welcome.

References

[Lazo86] E. D. Lazowska, J. Zahorjan, D. Cheriton, W. Zwaenepoel. File Access Performance of Diskless Workstations. *ACM Transactions on Computer Systems* 4(3):238-268, August, 1986.

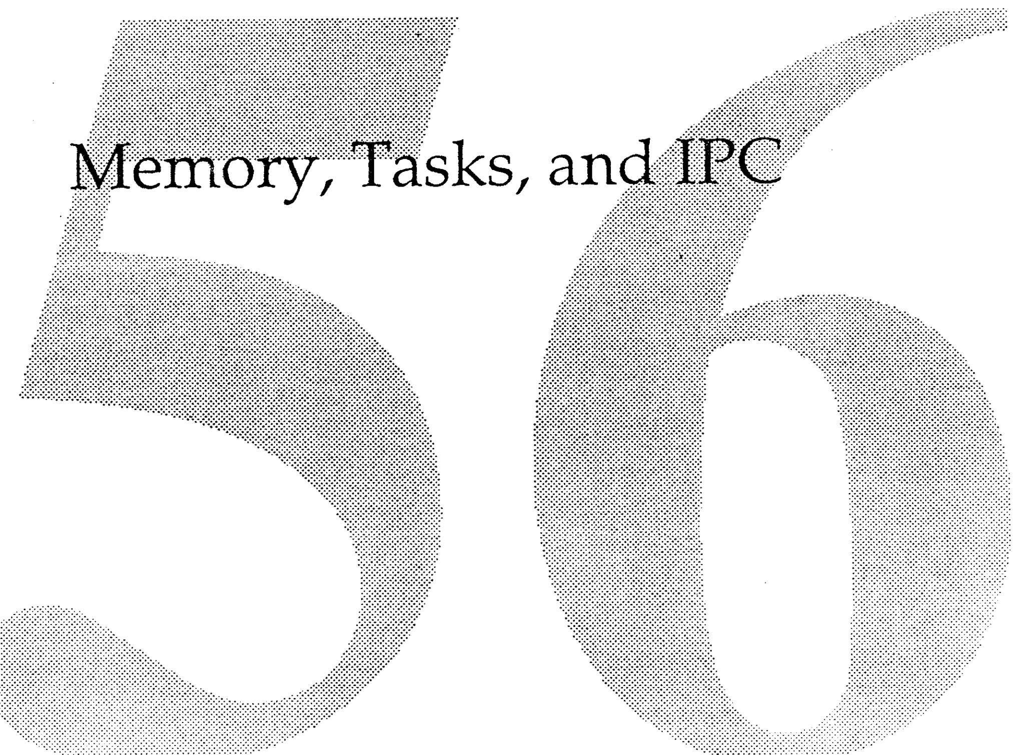
[Nels88] M. N. Nelson, B. B. Welch, J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):134-154, February, 1988.

[Saty85] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, M. J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th Symposium on Operating Systems Principles*. December 1-4, 1985. ACM, 25-34.

[Srin89] V. Srinivasan, J. C. Mogul. Spritely NFS: Experiments with and Implementation of Cache-Consistency Protocols. Research Report 89/5, Digital Equipment Corporation Western Research Laboratory. March, 1989.

[Venk89] R. C. Venkatraman, R. Andrews, G. Burns, M. Day, C. Kim, G. Vaughan. AFP++. Internal Document, Fifth Revision, March 1989.

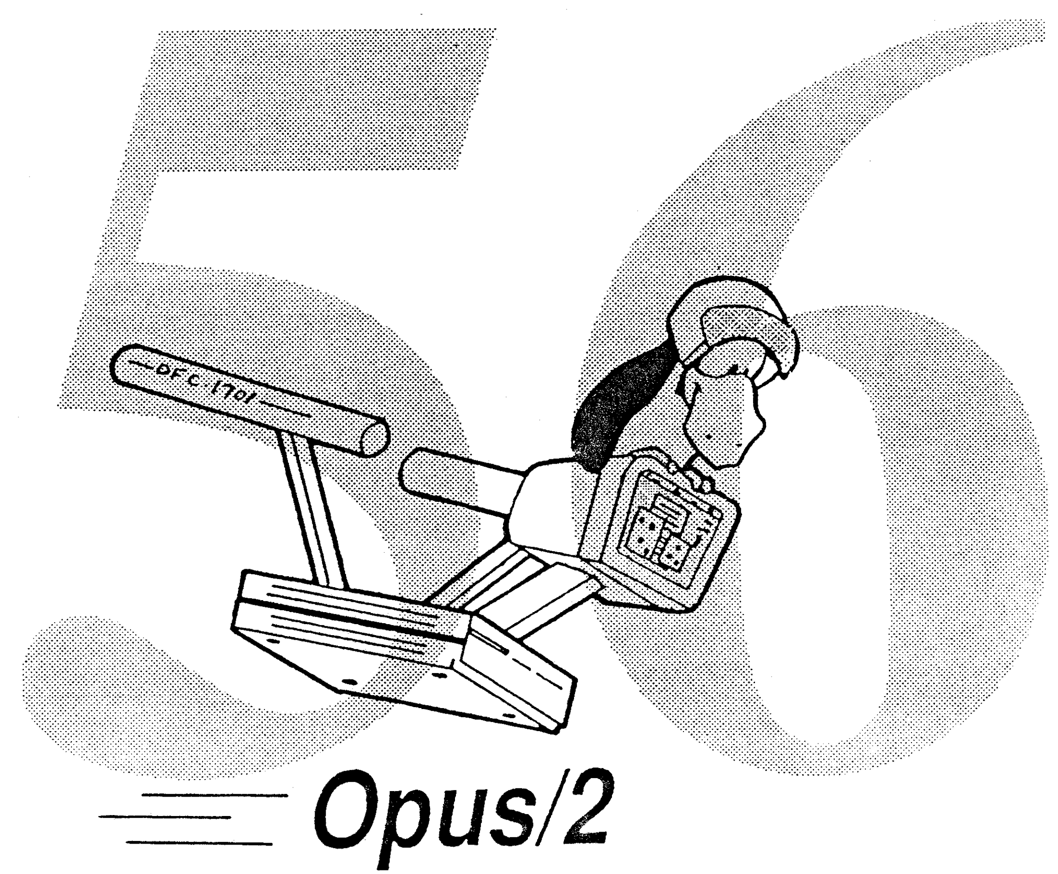
Opus/2


Memory, Tasks, and IPC
56

56

Memory, Tasks, and IPC

Lee Bolton, Dan Chernikoff, Chris McFall, Chris Moeller, Deb Orton



56

Overview of Kernel Services

The kernel provides a basic set of services for its clients based on a few key abstractions. The abstractions and the general nature of the services available for each are described briefly below. Kernel services always have a defined scope within which they are valid. In this document we distinguish between *local* services, which execute on a single kernel and *remote* services, which can execute on different kernels separated by a bus. All kernel services are encapsulated in objects, which are either "real" or "surrogate". Real objects represent entities or services that a task or team has directly invoked; surrogate objects represent entities or services on other tasks or teams.

Address Spaces and Memory

An address space is the "world" within which a program executes. More formally, it is the set of all possible addresses that the program can generate and try to reference. With an MMU, an address space can be built that is protected, thus providing a private, isolated environment in which the program can run. The kernel allocates and assigns machine memory resources to an address space as required by the program during its execution.

An address space is composed of several segments that are used by the program for various purposes: program code, shared library code, global data, stack, etc. Additional segments can be established by the program dynamically for its own private use or for sharing data with programs in other address spaces. These segments are similar in nature to the shared memory segments of UNIX or the memory-mapped files of TENEX or Mach.

The memory management interfaces of the kernel allow programs to create new segments in the address space, to open existing segments, to change their size, to get information about them, and to close and remove segments from the address space. Additional services allow control operations on portions of the address space (e.g., flushing to backing store, locking in physical memory, etc.) and advisory operations (e.g., will reference, won't reference). The scope of the memory management services is both local and remote – segments are generally managed in the local domain, but shared segments can be accessed in the remote domain also.

Tasks and Teams

A task is an execution entity in an address space. Put another way, a task is a thread of execution. Tasks are the entities in the system that accomplish work; they are the units of scheduling and execution in the system. Traditional operating systems such as UNIX allow only one task to be executing in an address space at a time; in fact, the task and the address space are synonymous. It should be possible, however, to have more than one thread of execution in an address space at any given time. Having multiple tasks executing in a common address space has several advantages: sharing data is easy, context switch time can be reduced, and task creation time can be faster. The collection of multiple tasks in an address space is called a team, because the tasks will generally be cooperating in a concurrent fashion to implement an application or service.

The process management interfaces of the kernel allow programs to create multiple tasks in an address space, to create entirely new teams of tasks in new address spaces, to get information about tasks and teams, and to destroy tasks or entire teams dynamically. Additional services allow tasks to adjust scheduling priorities to reflect the urgency they have in execution. The scope of the process

management services is local – tasks and teams are managed only in the local domain.

Interprocess Communication

In order for tasks on a team or for tasks on different teams to work together, some form of communication must be possible. The shared address space of tasks on a single team provides one means for tasks to communicate, but tasks executing concurrently or in parallel will generally need to synchronize their activities as well as communicate. Interprocess communication allows tasks to send and receive messages in a controlled and synchronized fashion. IPC models vary widely among different operating systems.

The kernel provides a synchronous model of IPC with short, fixed-length messages. A task can send a message to another task and is automatically blocked until the receiving task has sent a reply message. Such a send–receive–reply sequence is known as a transaction; transactions have easily understood semantics, which will encourage the use of multiple, concurrent tasks in building applications. The kernel also provides a set of asynchronous IPC services that allows tasks to send and receive messages without blocking. In addition to the message passing services, the kernel provides a bulk data transfer facility for efficiently moving large amounts of data between address spaces (and tasks).

The interprocess communication interfaces of the kernel allow a task to synchronously or asynchronously send a message to another task, to receive a message from any task or from a specific task, to reply to the originator of a transaction, and to forward a message to another task for processing. Additional services allow a task with proper rights to read or write data in another task's address space. The scope of the IPC services is local and remote – tasks can communicate with each other transparently irrespective of the domains in which they are executing.

Exception Handling

One of the goals of Pink is to allow developers to create robust applications that correctly handle error conditions. The operating system (kernel and runtime) allows the programmer to control what action to take when such conditions occur. It provides a uniform model for exception handling to the application programmer for both hardware- and software-detected errors.

The software exception handling model supported by the runtime system is based on the exception model provided by the C++ language. The hardware exception handling model provided by the kernel is based on tasks and IPC. By packaging a hardware exception as an IPC transaction with a handling task, an easily understood mechanism for dealing with hardware exceptions is achieved.

The hardware exception handling interfaces of the kernel in particular allow a task to identify itself as a hardware exception handler for a team and to get and set task state information as a means of handling exceptional conditions. The scope of the hardware exception handling services is local – hardware exception handling tasks must be local to the kernel of the task incurring a hardware exception.

Format of this document

The following four sections of this document present the services available in each of the major areas highlighted above. Each section follows the same general format. A section begins with a

presentation of the terminology associated with the area being discussed. This provides a common ground for the remainder of the discussion and ensures that the reader has a clear idea of what the terms are being used to convey. A section then continues with a detailed description of the model for the services of the area being presented. A section ends with diagrams of the class hierarchies involved in the particular area being presented.

Much of the work in this specification is based on several other systems described in the literature. In particular, the Thoth system from Waterloo, the V-Kernel from Stanford, the XMS system from BNR, and the Mach system from Carnegie-Mellon have had a great influence on the shape of this design.

Memory Management

Terminology

Memory management refers to a mechanism in which the stream of logical addresses generated by the processor is translated into physical addresses, which can be used to access real memory. The *logical address space* of a task (executing program) is the set of addresses that it can legitimately access. The *physical address space* of the system is the set of addresses that can be used to access the RAM that exists in the machine; the physical address space is also sometimes used to control the I/O devices connected to the machine. The memory management mechanism is provided by a hardware *Memory Management Unit (MMU).* The major function of an MMU is to implement the translation operation described above. Along with address translation, the MMU can perform certain checks on the type of memory access being requested by the processor. These checks can implement different kinds of protection for the user programs and operating systems that may be executing.

A *virtual memory* system is one in which a logical address space, generally larger than the physical memory present in the system, is provided automatically to application tasks by the operating system. In such a system, the logical address space of a task is sometimes called the *virtual address space,* and the physical address space of the system is sometimes called *real memory.* Since the virtual space is larger than the real memory in the system, the operating system can only allow pieces of a task's address space to be kept in memory at a time. As new pieces of the address space are required and referenced by a task, the operating system automatically fetches them from a backing store (e.g., a disk) that contains a complete copy of the address space.

In a *paging* virtual memory system, the logical address space is divided into a (large) number of fixed-length units called *pages.* Sets of pages generally correspond to programmer-defined portions of the address space (e.g., stack, heap, program code, library code). Pages are the unit of transfer between backing store and real memory in a virtual memory system. The paged address space is usually managed in one of two ways. The first way is to treat the address space as one large linear progression of bytes. Programmer-defined objects are then located at various positions in the address space. The second way to manage a paged space is to consider the address space to have a higher level of segmentation placed on top of the pages. This method is usually referred to as *paging beneath segmentation.* Subsequently in this document, references to paging are intended to include both methods.

There are several page-based MMU's available as separate parts or integrated with processors that are very cost-effective. These MMU's typically have an address translation cache that contains some number of the most recently used virtual-to-real translations. If a translation is needed that is not in the cache, page tables in memory must be searched, and the necessary entry loaded into the cache. This searching is either done explicitly by software after a signal from the MMU, or it is done automatically by the MMU itself. Strictly speaking, context switch time is very low because only a single MMU register (a pointer to the page table) must be loaded. However, the true overhead for this kind of translation process is difficult to measure. After a context switch, a task will incur a number of translation cache misses until its locality is effectively covered again.

Memory Management Services

The set of memory management services provided by the kernel must be able to support the range of possible MMU and system architectures that will exist in the Macintosh family of machines. In all of the memory architectures currently envisioned, the MMU provides a page-based model for the

address space. The MMU's differ mainly in their specific parameters and their specific operations. Examples of differences in parameters include page size and structure of the address space (e.g., number of levels and granularity at each level); examples of differences in operations include translation cache flushing and address validation. For details on a specific MMU, such as the Motorola MC68030 or MC88200, the reader should consult the appropriate reference manual. The memory management services provided by the kernel are defined to operate in the local and remote domains – memory can be managed and controlled within a single kernel or between kernels on a bus.

Kernel Memory Management Model

The address space of a task is the set of all addresses that the task could attempt to access. On all currently envisioned architectures, the address space is accessed by 32-bit addresses and is four Gigabytes in size. The amount of that space available for user applications varies with the particular MMU, but is generally several Gigabytes.

A *segment* (TSegment or TSurrogateSegment) is a named contiguous region of virtual memory that can be mapped into a task's address space. The name space for segment names is the same as the name space of the Pink file system. Using a single name space allows segments to be shared by several tasks as needed, and it allows segments to have a permanent storage location if desired (TFileSegment). When mapped into a task's address space, a segment has a starting address, a size, and several attributes. The starting address of a segment must always be on a page boundary, and its size is always a multiple of the system page size. The location of a segment in the task's address space can be selected by the kernel or specified explicitly by the programmer. Some implementations may have further restrictions on where segments can be located (e.g., on a 16 MByte boundary), so that the programmer may have to be more cognizant of the underlying MMU architecture when explicitly locating segments in an address space.

A segment is mapped into a task's address space by being either created or opened: new segments are created, and existing segments are opened. A segment can be created as temporary or permanent. A temporary segment is deleted from the system after it is closed; a permanent one remains in the system (usually on backing store) after it is closed and can be opened and accessed again later. A segment can also be created as memory resident or with backing store. A memory-resident segment has all of its pages in real memory whether it is mapped by a task or not. A segment with backing store is generally subject to the normal paging operations of the virtual memory manager.

A segment can be created or opened for private access or shared access. A private segment is accessible only to the address space in which it is created or opened; a shared segment is accessible across multiple address spaces (TSharedSegment). Subclasses of the shared segment are provided (TServerSegment, TClientSegment) to support a "server/client" model of shared memory: the server creates the segment and allows clients to open and use the segment A segment can also be created or opened so that it can be accessed in a read-only or read-write fashion. An attempt to do a memory write in a read-only segment will generate an access fault hardware exception in the offending task; a read-write segment can be referenced by both memory reads and writes. In some implementations, it may be possible for different tasks to map the same shared segment with different characteristics in their address spaces as they choose (e.g., at different locations with different sizes and different access permissions).

When a segment is mapped into a task's address space, a segment identifier is returned in an object (TSegment). This object containing the identifier can be used to refer to the segment in other kernel interfaces. Note that the segment identifier, and therefore the segment object, only has meaning within the team that makes the call mapping the segment.

A task can only legitimately access those portions of its address space where segments have been created or opened; attempts to reference memory where a segment is not defined are improper. With an MMU, an improper memory reference will cause an hardware exception, which the kernel will handle as needed. Such a hardware exception can either be forwarded to some other task for handling, or it can cause the task generating the exception to be terminated.

The pages of a newly created segment are zero-filled when they are first accessed. When a segment is subsequently opened, the specified size may be different than the size of the backing store associated with the segment. If the specified size is smaller than the backing store, only that portion of the backing store covered by the size of the segment in the address space can be affected by memory read and write operations. If the specified size of the segment is greater than the backing store, additional pages are zero-filled and allocated to the backing store as they are referenced.

When a task begins execution, the kernel will initialize its address space with several pre-defined segments (e.g., a stack segment, a global data segment, and one or more code segments). Additional segments may be set up by the run-time environment that a particular client is using (e.g., multiple segments for shared library code and data). There is no kernel constraint on the relationship between segments in the address space, other than the restriction that they may not overlap one another. Conventions may be established in the future by the various run-time environments for the locations of the segments that they manage. For example, the segments used to provide shared library code may be placed close together to conserve virtual address space for other uses.

Additional segments can be created or opened and subsequently closed by a task dynamically as it executes. These programmer-defined segments can be placed anywhere in the address space that does not already contain a segment. After a named segment has been created by a task, it can be opened and shared by other tasks. Segments thus integrate several memory management objects into a single unified concept: they can be used for private, temporary memory (TSegment); they can be used to share memory (TSharedSegment, TServerSegment, TClientSegment) between tasks in a fashion similar to the shared memory facility of UNIX System V; and they can provide the basic functionality and behavior of memory-mapped files (TFileSegment) found in some operating systems such as TENEX, Multics, and Mach.

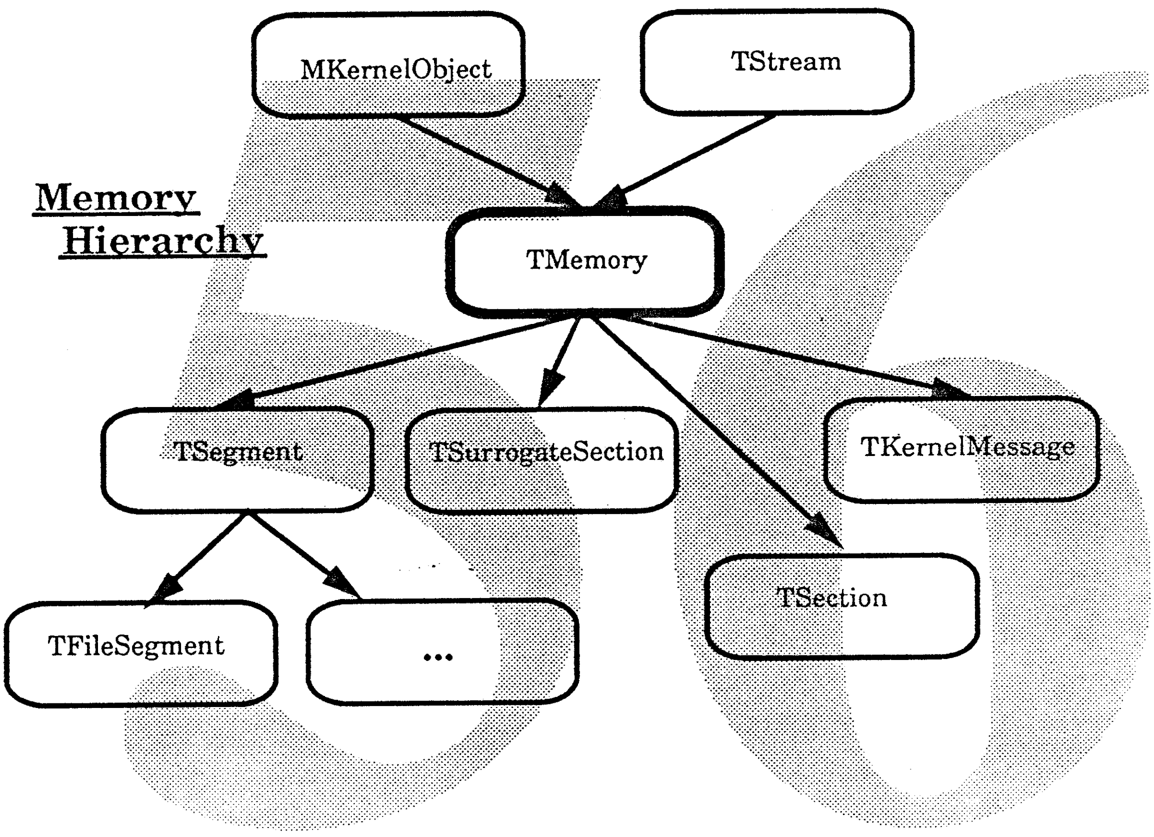
A segment normally has backing store associated with it that is located on some block-structured device connected to the system. This backing store is mapped into the address space when the segment is opened. A *physical segment* (TPhysicalSegment) is one in which the backing store does not correspond to such a device, but rather to some other physical resource in the system, such as the bitmap display or the I/O registers of a device. As with a normal segment, a physical segment can be mapped anywhere in the virtual address space. References to these pages in the virtual address space are mapped directly to the physical addresses associated with the segment and affect the associated physical resource in a hardware-specific fashion.

The virtual address spaces of all the tasks running in the system will generally exceed the amount of real memory available on the machine. The virtual memory manager will thus treat the physical memory as a cache for the most frequently used pages of the executing tasks. General page placement and replacement algorithms attempt to approximate this set of pages by observing the referencing patterns of the running tasks. The approximation, however, can sometimes lag behind the executing tasks, causing additional page faults. This lagging can often be mitigated through the *memory management advice* facility, in which a task informs the kernel of its intended use of the address space (TMemory). In this way, the memory manager can often have the data required by a task in real memory before actually being faulted for it by the task.

In a virtual memory environment, it is sometimes necessary for certain clients to have more explicit control over portions of the virtual address space, especially with respect to the paging that is

normally transparent to applications. A service is provided that allows control operations to be applied to sets of pages in the address space (TMemory). These operations include locking a set of pages in memory, preventing them from being paged; freezing a set of pages in memory, preventing them from moving; flushing a set of pages to backing store, ensuring consistency of data; and changing the access protection (i.e., read-only or read-write) on a set of pages.

Memory Management Classes



Process Management

Terminology

A process is an instance of the execution of a program. A process can be thought of as an activity that is underway. Every process has an associated identifier, state, priority, and context (processor and memory management state information). *Multitasking* refers to a system that allows multiple processes to be underway at the same time. In the strictest sense, the term multitasking implies nothing about the management of physical memory. A multitasking system can be built that allows only one process to reside in memory at a given time, with a process swap being performed on every context switch. Informally, multitasking is often equated with *multiprogramming*, which refers to a system that allows multiple programs to be co-resident in memory, either wholly or partially, as a way to increase processor utilization by always having some process ready to execute. The term *multiprocessing* refers to a system that supports multiple CPU's.

A *lightweight process* is a process that shares state information – such as address space mapping, kernel resources, etc. – with a number of other, also lightweight, processes. Since there is little private state information associated with it, a lightweight process can be created and destroyed rapidly, and a context switch from one lightweight process to another can be less expensive than one between non-lightweight processes. A *team* is the collection of lightweight processes that share a particular set of state information. The team is the functional unit that can be thought of as owning the address space, access permissions, and resources acquired by the various lightweight processes on the team. For the remainder of this document, lightweight processes will be referred to as tasks. A newly created team is populated with one task, known as the *root task* of the team. The new root task is referred to as the *child* of the creating, or *parent*, root task. Although any task on a team can create a new team, the parent of the new team is considered to be the root task of the creating team.

The *team address space* is the set of addresses that is reserved for the use of the tasks on a team. With memory management hardware, the team address space will be a logical address space that is contiguous and usually larger than the real memory available on the machine. The memory management services of the kernel generally provide virtual memory mechanisms to the tasks on a team to efficiently manage the team address space.

Tasks are scheduled by the operating system for execution on a processor. A *non-preemptive* scheduling discipline allows a task to execute until it voluntarily releases the processor. The points at which the processor can be reassigned are always well-defined. A *preemptive* scheduling discipline provides for the involuntary removal of a task from a processor. By reallocating the processor at regular intervals, preemption can be used as a mechanism to provide fair sharing of the processor. This approach is known as *time-slicing*. Preemption can also be used as a mechanism for guaranteeing responsiveness, by reassigning the processor whenever a task of higher priority than the current one becomes ready to execute.

Process Management Services

The process management services are based on the model of lightweight tasks on a team. The tasks on a team generally cooperate in the performance of some activity. This type of model seems particularly appropriate to the kinds of hardware architectures and client programs that are anticipated in the future. The process management services provided by the kernel are defined to operate in the local domain – tasks and teams are managed and controlled within a single kernel.

Process Management Model

Creation of a new team requires the creation and initialization of an address space for the team. As part of this initialization, several segments are created or opened in the address space. These include a segment for the program code being executed by the team, one or more segments for the shared library code and data that the program may require, a segment for the static global data of the program, and a segment for the stack of the root task. All other portions of the team address space are initially undefined.

The set of all team root tasks is organized hierarchically according to the parent-child relationship. The individual tasks within a team are also organized hierarchically according to creation. The destruction of a task on a team causes the destruction of all subordinate tasks on the team. A team is destroyed when the root task of the team is destroyed. The destruction of a root task brings about the destruction of all root tasks, and hence all teams, descendent from that task.

A task is created in the blocked state, awaiting a reply message from the creating task. The new task will begin executing when the creating task replies, passing an initial message, known as the *start-up message*, which the new task can access as a parameter variable in the topmost procedure. The start-up message is uninterpreted by the operating system and can be used to pass any information from the parent to the child. In addition, since the child is created awaiting a reply message and with its address space open, the creating task can use the interprocess communication services to write data into the address space of the new team. The interprocess communication facilities are fully described in the Interprocess Communication Services section.

The basic task object (MBaseTask) is subclassed to provide objects for creating a new task (MTask) and representing an existing task (TSurrogateTask). A mechanism is provided for starting a task as a new team (using MTask and TTeam), and a subclass exists (MMessageTask) that provides methods to handle commonly received messages (e.g., alarms, death messages). The basic team object (MBaseTeam) is subclassed to provide objects for starting new teams (TTeam) and objects for representing a team that already exists (TSurrogateTeam). Because a team is designated by the root task of the team, all team objects descend from the basic task object (MBaseTask).

A team owns various resources that have been acquired through requests to the kernel. Examples of such resources are the address space and real memory. Tasks themselves do not own most resources, but rather share the resources owned by the team. One exception is a task's stack, which is owned by the task. A task's stack is allocated when the task is created and reclaimed when the task terminates. The stack is initially small and can be occasionally overflowed in the course of execution. When this happens the operating system automatically allocates the additional memory required to accommodate the stack. A task's stack is not allowed to grow beyond the size specified when the task is created. A task is aborted if additional stack space can not be allocated when required.

Tasks can request resources from the operating system, but those resources are considered to be owned by the team. When a task is destroyed, resources acquired by that task, such as open segments, are not reclaimed, but remain the shared property of the team. Since major resources need not be allocated or reclaimed, the creation and destruction of individual tasks is quite fast. However, because the destruction of a task causes the destruction of all tasks in its sub-tree, the overall task destruction time is a function of the complexity of the hierarchy beneath it. Also if the task has allocated some team global resources those resources can be lost if the task dies without releasing them.

Scheduling of tasks for processor resources is preemptive and based on the priority ordering of eligible tasks. The tasks running on the available processors are always the highest priority tasks that are ready to execute. A priority is an integer in the range 0 to 255, with 255 being the lowest, least

urgent priority, and 0 being the highest, most urgent priority. Every team has a *team base priority*, and each task on the team has a *relative priority* with respect to the team base priority. The absolute priority of a task is calculated by adding the task's relative priority to the team base priority. The relative priority of a task can be positive or negative. The team base priority reflects the urgency of a team with respect to other teams in the system; the task relative priority reflects the priority of a task with respect to other tasks on the same team. Figure 2.9.1-1 shows an example of task and team priorities. Note that it is possible for tasks on different teams to have absolute priorities that overlap.

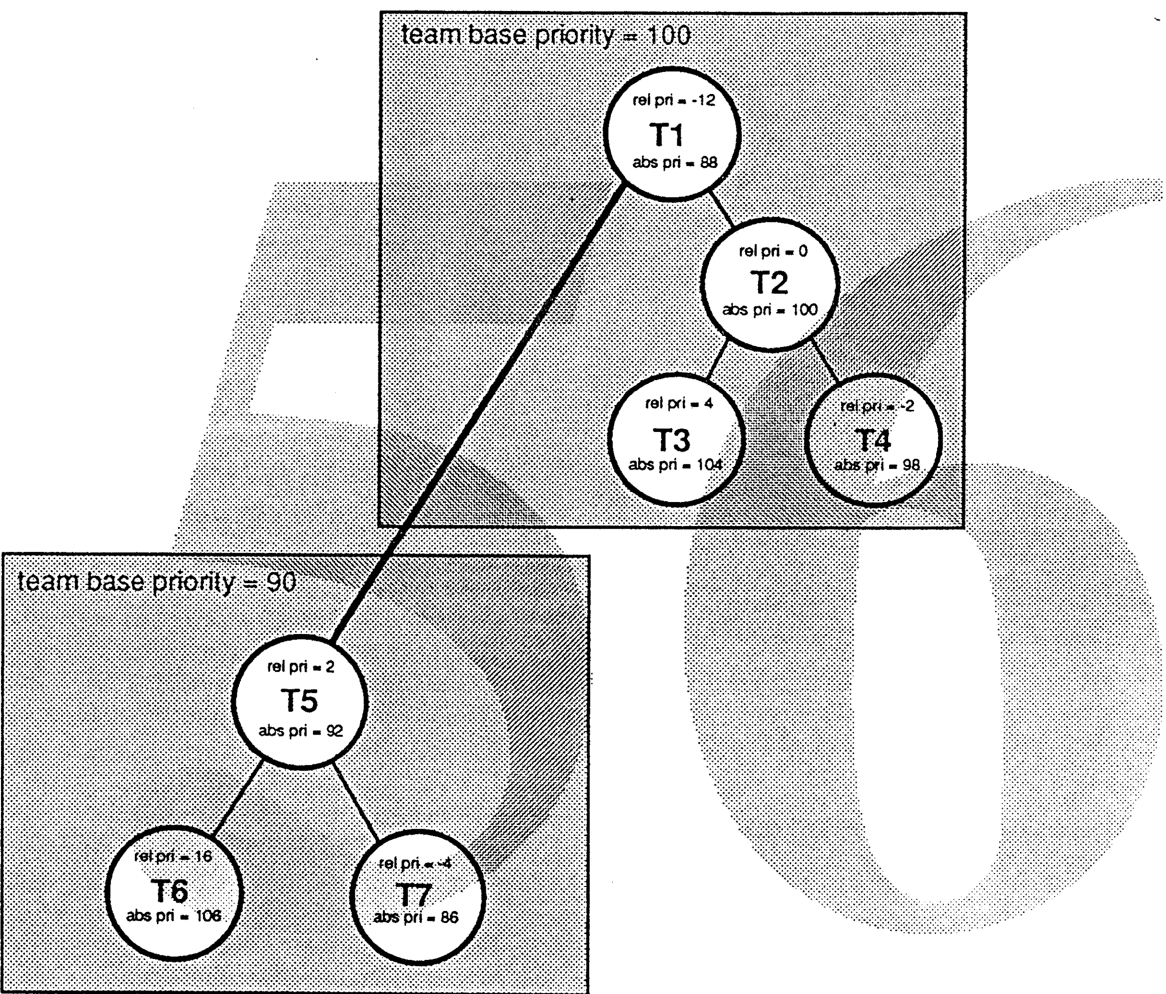


Figure 2.9.1-1. Task and Team Priorities

The assignment of task priorities should not be used by programmers as a mechanism to serialize and synchronize the execution of tasks. Interprocess communication or some other explicit synchronization service should be used instead. A task's priority represents the importance of the work being done by the task in relation to other work going on in the system; it can be thought of as a reflection of the percentage of processor cycles that should be allotted to the task.

A subset of the highest priorities will be reserved for use by the kernel and operating system. Some portion of the remaining priorities may be reserved for a class of scheduling in which execution is time-sliced to equitably share the processor among compute-bound tasks. Still another portion of the remaining priorities may be reserved for scheduling tasks with "real-time" requirements. While the

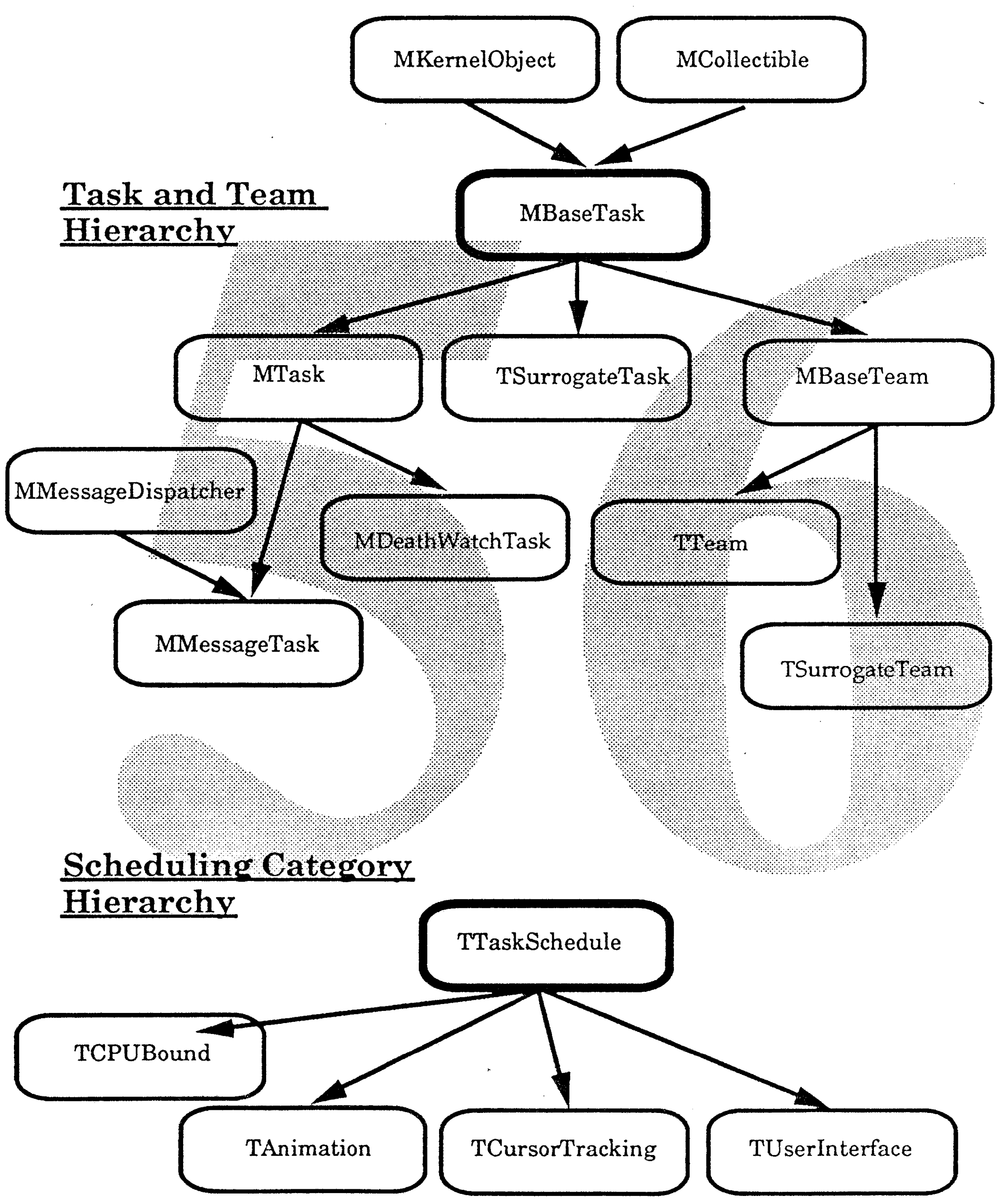
kernel currently provides preemptive priority-based scheduling as its only scheduling policy, other policies (e.g., deadline or time-slot) can be implemented through servers that provide whatever model is desired by the various application clients of the system.

Currently, the scheduling urgency of a given task is specified by an object (subclass of TTaskSchedule) representing the category to which the task should belong. These categories encapsulate the relative and base priorities described above. Priorities may not be set explicitly, but only through the use of task scheduling objects. Possible categories include *animation* (TTaskSchdeule::kAnimationTask), *user interface* (TTaskSchedule::kUserInterfaceTask), *mouse/cursor tracking* (TTaskSchedule::kCursorTrackingTask), *CPU-bound* (TTaskSchedule::kCPUBoundTask), and *servers* (TTaskSchdeule::kServerTask).

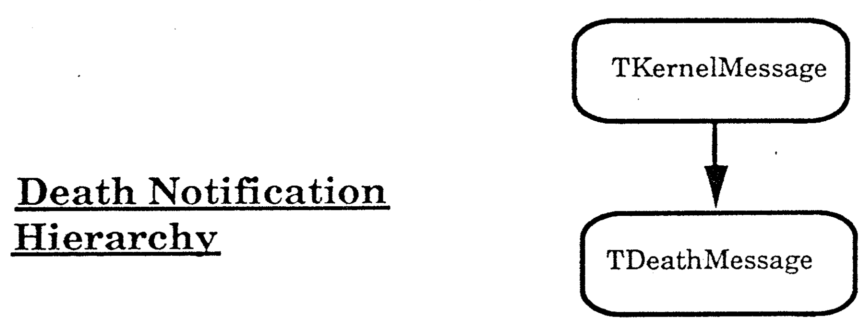
The tasks on a team cooperate in the performance of some activity. Since the team address space is shared among several tasks, synchronization of accesses to data structures becomes essential. Management of a shared team heap is an example of a facility that cannot be implemented with unsynchronized library routines. Synchronization should be accomplished using the message passing services of the operating system, which are described in the Interprocess Communication Services section or with the use of semaphores described in the *Runtime* document.

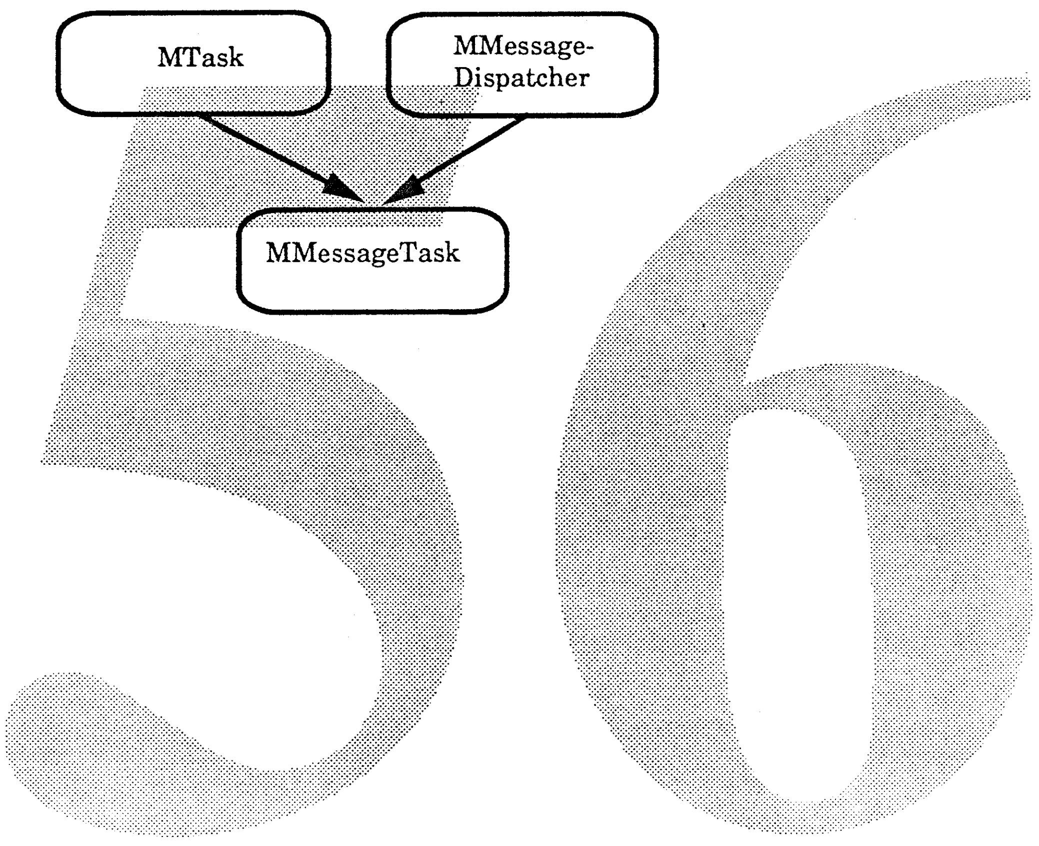
In teams that contain multiple tasks, graceful shutdown may be complex when shared resources are involved. To facilitate the graceful shutting down of tasks, and to provide notification of a task's death, three objects are defined. A death notification task (MDeathWatchTask) blocks until the specified task dies and then performs some cleanup operation. A death message (TDeathMessage) is provided that enables a task to block awaiting another task's death. A light-weight task (MMessageTask) is provided that sits in a loop dispatching messages and dies gracefully if a death message is received. A polling mechanism is also available, which reports whether or not a given task exists.

Process Management Classes



Death Notification Hierarchy





Interprocess Communication

Terminology

The kernel provides a task model based on concurrently executing lightweight tasks. In order for these tasks to work together, a mechanism must be provided to allow communication and synchronization between them. *Communication* means passing information from one task to another, either in the form of short control messages or of large chunks of data or address space. *Synchronization* allows two or more tasks to access shared resources (such as shared memory, CPU cycles, I/O, etc.) safely and in a controlled, predictable manner. This function of communication and synchronization between execution entities is commonly called *Interprocess Communication*, abbreviated *IPC*. We will adhere to this nomenclature to be consistent with the rest of the literature, even though it is tempting when talking about the kernel to call it *ITC* (Inter*task* Communication).

IPC can exist not only between lightweight tasks in the same address space, but also between tasks in different address spaces on the same machine, and even between tasks on different machines connected via a network. If a programming model is structured such that a task can not or need not tell whether the task it's communicating with is on the same machine or on a remote machine, then that model is said to provide *network transparent* IPC.

There are many different styles of IPC, differing in their underlying implementation as well as in the model they provide to the programmer. If a task can transfer a collection of information (called a *message*) to another task and immediately continue running, then it has an *asynchronous* message passing model. Conversely, if the task must suspend execution until the destination task has received the message (and perhaps done some processing on it) then it is using *synchronous* message passing. Asynchronous message passing typically requires the operating system to implement *buffering* and *queuing* of messages; that is, to store incoming messages as they are received and present them to the destination task as they are requested. If message queuing is provided, it may be important for the system to ensure that the messages are handed to the destination task in the order they are received – this is called *sequencing* of messages. It may also be desirable for the application to be able to specify the order in which the messages are received (i.e., the order in which they are placed in the task's incoming message queue). This is usually done by assigning a *priority* to the message.

An IPC model must provide not only the actual mechanisms for passing messages and synchronizing tasks, but also must define a *naming scheme* so that tasks can be identified. A naming scheme is basically a set of rules describing what a task's name is composed of, what names are valid, and how to relate a name to a specific task and vice versa. The set of all possible valid names under a given naming scheme is called a *name space*. An IPC model may require more than one name space – for example, it may need a name space to identify tasks locally and another to identify tasks uniquely across a group of machines networked together.

When talking about providing a *task identifier* to name a task, issues of uniqueness must be addressed. If task identifiers are integers assigned in ascending order and represented by a fixed number of bits, then at some point in time a maximum value will be reached, and the task identifiers will start being assigned from the beginning values again. This could cause identity problems, so the kernel must be careful to insure that a task identifier is not reused too quickly after its last use. For any given name space, the period of time between reuse of an identifier is called its *T-stability* (from David Cheriton's V-kernel work). Obviously, it is desirable to have as large a T-stable period as possible – preferably on the order of years, not hours.

IPC between two tasks can sometimes be categorized as a client-server relationship. The *client* task requests some action or service from a *server* task. The server task satisfies the request, sometimes responding with information or a status message. This exchange is called a *transaction*. If the response gets lost somehow (e.g., if the client and server are separated by an unreliable network), it may be possible for the client to just reissue the request, provided no ill effects will arise if the server performs the same request more than once. Such a reissuable request is termed *idempotent*.

Interprocess Communication Services

The purpose of kernel IPC is to provide synchronization and control between lightweight tasks within a single address space or across multiple address spaces. It is a programming tool (in the same sense that a subroutine call is a programming tool) used to resolve the issues of concurrency and synchronization between execution entities in a multitasking application. Network communication is (currently) provided at a higher level. The kernel programming model provides concurrency via multiple tasks running in one or more address spaces. Since the unit of memory protection is the address space, an application may want to extend itself across multiple address spaces to make use of that protection. An application may also want to extend itself across multiple CPU's to gain true parallelism or to take advantage of special hardware not available on the local CPU. In all these cases it is appropriate to use kernel IPC for synchronization and control. The scope of all kernel IPC services is defined to be the local and remote domains (i.e., between tasks on the same kernel or separated by a bus).

Overall IPC Model

The kernel interprocess communication model provides a set of very fast message-passing primitives for communication between tasks and for synchronization of the task operations (TKernelMessage). A kernel message has a fixed size and is composed of a *message header* section containing control and status information for the message transaction and a 128 byte *data* area for passing user-defined information. The kernel IPC model provides synchronous message passing for synchronization and control between tasks and asynchronous message passing for transfer of control messages and data in situations where the task doesn't want to immediately give up control of the CPU (i.e., doesn't want to block). Tasks directly name each other by task identifier when passing messages. A task may specify to receive a message from a *particular* task or *any* task; that is, a task may establish a one-to-one or many-to-one (server-style) method of interaction with other tasks. There is no facility for one-to-many (broadcast) or many-to-many transactions.

In addition to the message-passing primitives, the kernel provides the means for local or remote tasks to move data between their virtual address spaces via *sections* (TSection and TSurrogateSection). Two types of sections are defined: *permanent*, which are created and destroyed by explicit calls, and *temporary*, which are specified implicitly in the message header and are automatically closed at the end of the transaction. Access to another task's (temporary or permanent) section is allowed only in the context of a transaction, also called a *rendezvous*. Message objects may be reused in subsequent transactions, but rendezvous-specific information for past transactions is lost.

All task objects contain task identifiers. Surrogate task objects are used to refer to other tasks via their task identifier.

Rendezvous

When a message is sent, a rendezvous is initiated between the sender and the receiver. The rendezvous is finished when the receiver replies to the message. Each rendezvous is given an identifier that is unique within the task identifier name domain and only meaningful in the interval between the send and reply and kept by the message. For synchronous rendezvous, the rendezvous identifier is the same as the sender's task identifier. For asynchronous rendezvous, it is a unique identifier for each send. A rendezvous defines a message, a temporary section (if any), legal access to permanent sections, a source task, and a target task. While two tasks are in rendezvous, the receiver can access the portions of the sender's address space that have been opened as permanent or temporary sections.

Sender Side

All sends initiate a rendezvous. When the send is synchronous the sender is blocked during the entire rendezvous. When an asynchronous send is done, a *rendezvous identifier* is created and stored in the message. The sender can continue to execute until it does a receive to obtain the reply. At the completion of the receive the message may be queried about the identity of the replier. An invalid replier task indicates that the target task either didn't exist or terminated prior to replying. The message type indicates that the message is a reply. The message keeps track of the rendezvous identifier.

Receiver Side

A surrogate task for the sender is available after a receive. In the case of a receive of a request, it is a surrogate for the sender; in the case of the receive of a reply, it is a surrogate for the replier. To reply to a send, the receiver does a Reply to the message, which contains the rendezvous identifier, and the kernel automatically routes the reply to the task that did the corresponding send.

Message Passing

A synchronous IPC transaction is provided using the Send/Receive/Reply calls. A pair of asynchronous IPC transactions is provided using AsyncSend/Receive/Reply/Receive. And a one-way asynchronous IPC transaction is provided using OneWaySend/Receive/Reply. All three sets of IPC interactions have been designed so that a server task can do a Receive followed by a Reply and not have to worry whether the client task used Send, AsyncSend, or OneWaySend. This should simplify the design of servers that want to be able to handle both synchronous and asynchronous requests. There is also an AsyncReceive call that implements a non-blocking receive facility, and a ReceiveSpecific call that allows the receiver to select which message to receive based on sender or rendezvous identifier. Both these receive calls can be used anywhere Receive can.

Synchronous (Blocking)

A synchronous rendezvous is initiated using Send. Send suspends the task until the message is received and replied to. A sending task in this state is said to be *send-blocked*. A task receiving a message using Receive or ReceiveSpecific is suspended until a message is available. A task in this state is said to be *receive-blocked*. The two communicating tasks will synchronize their execution, or rendezvous, in order to pass a message. During the rendezvous, the message is copied from the sending task's data area into the receiving task's data area without intermediate buffering. The

receiving task becomes ready, but the sending task remains suspended until the receiving task sends a corresponding reply message using Reply. A sending task in this state is said to be *awaiting reply.* If a task has more than one task send-blocked on it when it does a receive, it will rendezvous with the task whose message has the most urgent priority (priority is specified in the message header). When a task dies, all other tasks that are blocked on that task (i.e., receive-blocked, send-blocked, or awaiting reply) are unblocked and allowed to execute. An aborted ReceiveSpecific or Send will generate a toolbox exception. Figure 2.9.1-2 below illustrates a synchronous IPC rendezvous. The shaded portion of each task arrow indicates the time period the task is blocked.

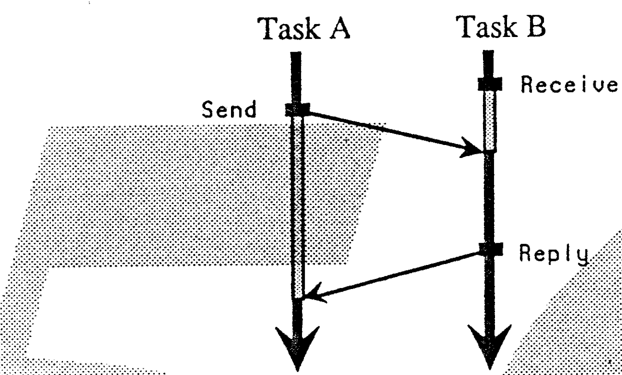


Figure 2.9.1-2. Synchronous Rendezvous

Asynchronous (Non-Blocking)

There are two styles of asynchronous transactions: a request-reply pair (initiated using AsyncSend), and a one-way send (using OneWaySend). When a task does an AsyncSend, a kernel message buffer is allocated to hold the message, and the buffer is queued onto the target task's incoming message queue in priority order (priorities are specified in the message header). The sending task does not block. When the target task does a receive, it gets the message buffer and the task identifier of the original sender as well as a rendezvous identifier (in the message header) that identifies this transaction. This rendezvous remains open (and thus section access is allowed) until the receiver does a Reply to the message (i.e., to a message buffer containing the rendezvous identifier in its header). This reply sends a message back to the originating task and terminates the rendezvous (additionally freeing up any kernel resources consumed by the rendezvous). The reply message is queued on the originating task's incoming message queue until the originator does a receive to pick up the message. If an AsyncSend is done to a non-existent task identifier (or the task dies before receiving the message from its incoming message queue), the kernel will reflect the message back to the originator, queuing it on the sender's incoming message queue. When the originator does a receive, the receiver task will be invalid. Figure 2.9.1-3 illustrates an asynchronous request-reply rendezvous.

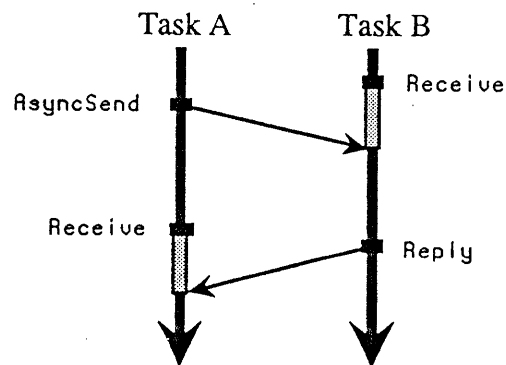


Figure 2.9.1-3. Request-Reply Pair Rendezvous

When a task does a OneWaySend, a kernel message buffer is allocated to hold the message, the buffer is placed onto the target task's incoming message queue in priority order, and the sending task is allowed to continue running, as in AsyncSend. When the target task does a receive, it gets the message. It should eventually do a Reply to the message, at which point the rendezvous is considered closed. The purpose of replying to a message sent via OneWaySend, however, is solely to close down the rendezvous. The message included in the Reply call is not returned to the originator. Figure 2.9.1-4 illustrates an asynchronous one-way rendezvous.

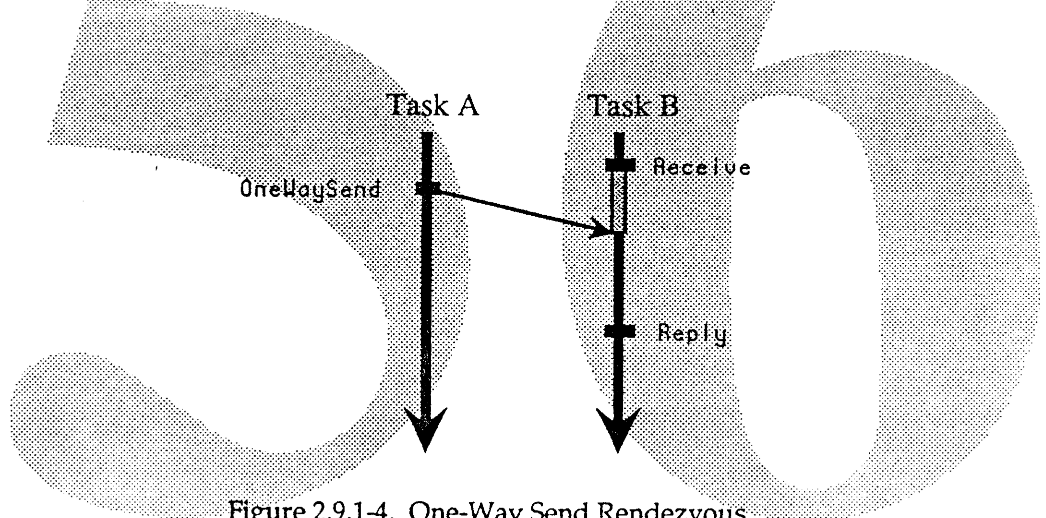


Figure 2.9.1-4. One-Way Send Rendezvous

There are two major differences between AsyncSend and OneWaySend. The first is that a OneWaySend is not guaranteed to be reliable: the sender receives no indication that the message was or was not delivered successfully. In the case of AsyncSend, the sender will receive a toolbox exception if delivery is not possible. The second difference is that while sections may be sent with both OneWaySends and AsyncSends, in the case of OneWaySends special handling occurs. For a OneWaySend the sender relinquishes ownership of the section, and deallocation of the memory is handled by the message object (after the receiver does a reply or immediately if the receiver no longer exists). Sections passed in a OneWaySend may not be referenced by the sending task after the OneWaySend is initiated. A light-weight task (TOneWaySendTask) is used to clean up the section associated with the OneWaySend and offers the convenience of sending a section and not having to worry about its deallocation.

Message Forwarding

A receiving task can use Forward to forward a message to another task, possibly altering the text of the message first. The effect of a forward is the same as if the sending task had originally sent to the forwarded-to task. A task can determine if the message just received was forwarded by issuing the Forwarder call. In the case of a message being forwarded more than once, the Forwarder is the last task to forward this call.

For example, in Figure 2.9.1-5 Task A sends message m to Task B, who forwards the message to Task C. Task C's receive call returns Task A as the sender, and a Forwarded call would return Task B as the forwarding task. When Task C does a reply, the reply message is sent to Task A, whose send returns Task C as the replier.

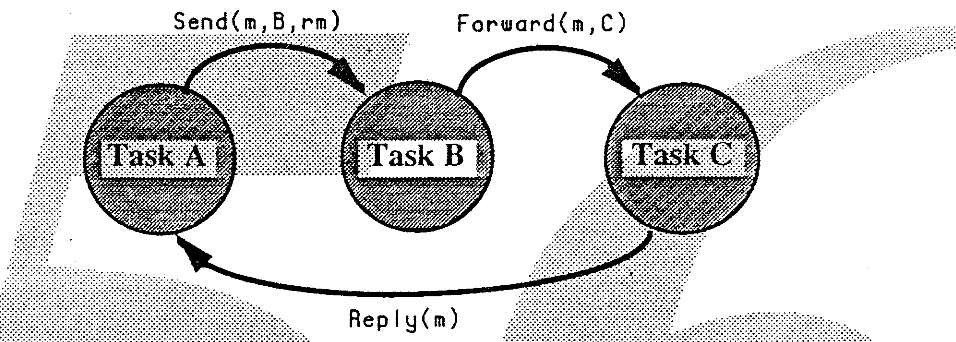


Figure 2.9.1-5. Example of Forwarding

Sections (Data Transfer)

The kernel message primitives provide fast, synchronized communication of short messages. A separate mechanism is provided for the inter-task transfer of larger quantities of information. Such data transfer occurs in the context of a rendezvous, when each of the communicating tasks is in a stable state. The data transfer model consists of a task "creating" a section (TSection) in its logical address space and doing a method call to associate the section with a message object (TKernelMessage). The term "section", as used in this discussion, refers simply to an extent of a task's address space and should not be confused with memory management segments. The section is made accessible to another task by entering into a rendezvous with it. Specific read and/or write access permissions can be granted to the receiving task. Data transfer is accomplished by the receiving task doing a method call to get a surrogate section object (TSurrogateSection) from the message, and then performing Reads and Writes to the surrogate or using the stream interface. Access to the section is terminated when the rendezvous ends. It should be noted that sections provide a mechanism for copying data between address spaces. To *share* data without copying, memory management segments should be used.

There are two kinds of sections: permanent and temporary. Both kinds can be accessed using Read and Write. *Permanent* sections are created by specifying "kPermanent" for the permanence parameter to the constructor, and can be accessed by any task that is in rendezvous with the task that the section belongs to, subject to the read/write permissions assigned to the section when it is created. *Temporary* sections are created by specifying "kTemporary" for the permanence parameter in the constructor and doing a Send, AsyncSend, or OneWaySend to another task. (Remember, in the case of a OneWaySend, the section is no longer available to the sending task!) The section is automatically closed when the rendezvous terminates and is accessible only by the task the caller is in rendezvous with. For all three types of sends, a rendezvous is defined to exist until the message is replied to.

A task may have multiple sections active (one temporary section and several permanent sections) at any given time. Permanent sections may not overlap each other. A temporary section may overlap a permanent section, in which case its access rights will take precedence. To facilitate connecting to systems with different hardware architectures (different byte orderings, size of addresses, byte versus word addressing, etc.), the standard stream operation Seek is used to position the stream at an offset to the beginning of the data, before subsequent reads and writes.

A temporary section is associated with a message (TKernelMessage), exists only during the lifetime of the message (send/receive/reply), and is identified by the message's rendezvous identifier (stored inside the message object). Each outstanding AsyncSend can have a temporary section defined. The receiving task may access the temporary section and any of the permanent sections defined by the originator of the message data through Read and Write (or stream operations) on section surrogates (TSurrogateSection). Note that to get the surrogate for a permanent section to the receiver, the sender typically streams the section object into the message sent to the receiver.

Message Priority

Messages have a *priority* associated with them that determines the order in which they are received, if a task has multiple messages in its message queue. The sender writes the desired priority into the priority field of the message header before sending the message. The most urgent priority has a value of -127, and the least urgent priority has a value of 127. A priority value of zero should be considered the baseline or "average" priority.

Naming

Communication paths between tasks are not formally protected. Presentation of a task identifier is taken as prima facie evidence by the kernel of authorization to send a message to that task. Where required, it is up to higher-level software to implement inter-task security.

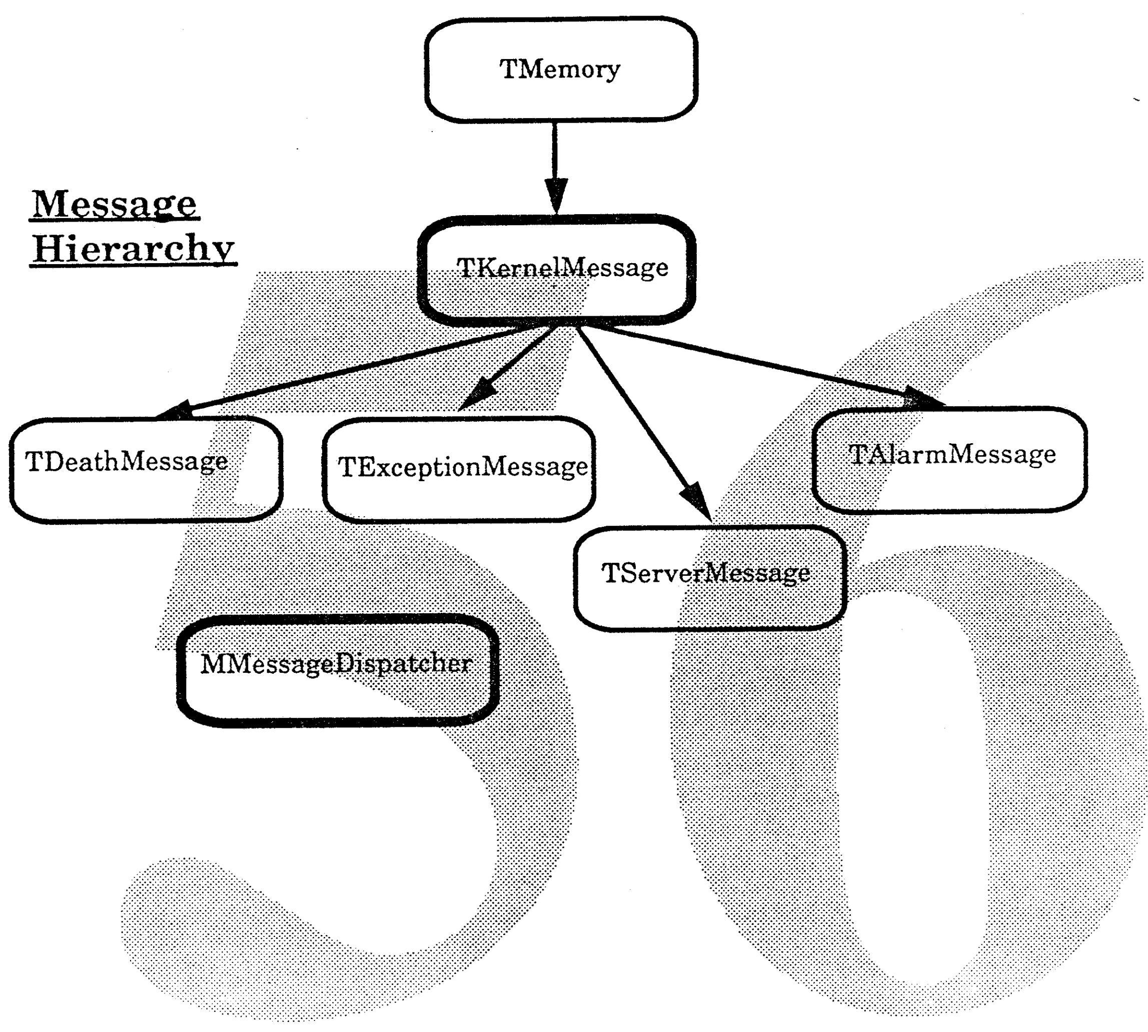
Kernel task identifiers are 64 bits long and are guaranteed to be unique within the kernel local and remote domains (i.e., for all tasks running on the same kernel and on kernels running on the same bus).

Task identifiers have an internal format that distinguishes between local and remote identifiers. The exact internal format of a task identifier may change depending on the version of the kernel and the underlying hardware. However, information about whether a task identifier is local or remote, and the values of the various components, will be available through library calls implemented above the kernel. In addition, the kernel guarantees that task identifiers will not be reused for a time interval on the order of years (i.e., the T-stability of kernel task identifiers exceeds one year).

Tasks make their task identifiers known to other tasks by registering a name with some name service. This name service is not implemented in the kernel and thus is not in the scope of this section. However, for bootstrapping purposes, the task identifier of the local name server will be made known to all tasks in the system in some manner.

56

Interprocess Communication Classes



Exception Handling

Terminology

A *software exception* is an error or status condition detected by program code, such as out of memory, file not found, and other application-specific conditions. A *hardware exception* is a hardware-related condition caused by the execution of a task, such as a floating point exception caused by dividing by zero, an addressing error caused by a pointer referencing an address outside the bounds of the team's address space, an illegal instruction, and so on. The default action in such cases is for the task to be terminated with an error indication. However, these errors often are recoverable. A means must be provided for the exception to be caught by a designated *handler* that can fix whatever went wrong and resume execution of the errant task, execute some clean-up code and kill the task, or provide debugging.

Exception Handling Services

A combination of the runtime system, MBaseTask, MBaseTeam, and exception handling classes provide the exception handling services. Exception handling operates independently in each task. Exceptions can be caught in the same task in which they occur or control can be referred to another task, such as a debugger. The runtime system (through C++ exceptions) handles the former, and the exception handling tasks handle the latter.

In addition to application programs, the exception mechanism must support other clients as well. Examples are garbage collectors and other software emulators, such as the Blue Adapter. These clients need to capture various hardware exceptions, such as ALine instructions and improper memory references, in a fashion transparent to the application.

Exception Handling Model

The exception handling model provides a uniform way to handle hardware- and software-detected errors. However, there is a natural tendency to handle each type of exception in a particular way. The same task that detects software errors, via the C++ exception mechanism, usually processes them. There can be deviations to this rule in that a programmer may choose, during development, to have all or some specific software exceptions sent to a debugging task. A designated exception handler task usually processes hardware exceptions. There are categories of exception types defined to help the programmer: individual (ErrorType), all software, all hardware, and all exceptions. The name space for exceptions includes all software and hardware exceptions. The default behavior of an uncaught exception is task termination.

Some exceptions can be designated as *conditional*, which means that if a handler is specified, the exception is sent to it. Otherwise, control returns in-line with no error handling. The only hardware exceptions that should be specified as conditional are those where returning control without modifying the task state cannot cause the exception to recur (e.g., Trap instructions).

Inline Exception Handling Model

Pink will implement the C++ exception mechanism when Bjarne Stroustrup's proposal is available. Both hardware and software exceptions can participate in the C++ mechanism and return control to the thread in-line. A task can specify that particular hardware exceptions are to be treated as C++ exceptions and can provide, optionally, a function to run to process the exception. The function can raise an exception or return. There is a default in-line function provided that will raise a C++ exception.

Task Exception Handling Model

A task can register itself as the exception handler for exceptions in another task or team. The MBaseTask and the MBaseTeam classes define methods for registering handlers, and the MExceptionHandler class defines methods for a task to register itself as a handler. The exception services use the client/server classes as well as the kernel task and IPC models. When a task incurs an exception that is to be handled by an exception handler, the exception mechanism causes a message to be sent from the task to an exception handler task. The message describes the exception that occurred and opens the address space of the task as a temporary section. The state of a task that is in exception processing can be viewed and modified by the task's exception handler only while they are in rendezvous. The exception handling services operate in the local domain – both tasks must be on the same kernel.

Associating a handler task with a team or task allows the user to capture and post-process exceptions in a separate task. An exception handler can register to receive one, some, or all exceptions from a task or team. For example, a debugger may want to see all exceptions from a certain team or task while a garbage collector may only be interested in access violations. A task becomes an exception handler task by making a RegisterAsHandler request or being specified as the handler in a RegisterForException request. The requests either specify the exceptions to be caught or 'all' for all exceptions. The last task to be registered takes precedence. When the current handler relinquishes control, by deregistering or terminating, the previous handler becomes the exception handler. If the handler registers as the handler for the team, it receives exceptions caused by any task on the team. An exception handler task should mix-in MBaseTask or MMessageDispatcher to get the necessary functionality. An MExceptionHandler task can register and deregister itself as the handler for a task or team for one, some, or all exception types. It can also specify which exception types are conditional in a task or team.

The handler calls Receive (or WaitForMessage if two or more message classes are to be received) to get the exception message (TExceptionMessage). The message identifies itself as an exception message, indicates the exception type, and specifies the task that was running when the exception occurred. The message also includes other information that may aid in recovering from the exception (e.g., the accessed address, the value of the program counter, etc.). The handler task can repeatedly do WaitForMessage and Reply calls to handle multiple exceptions. If the exception handler doesn't want to process the exception, it can do a Refer call to cause the exception to be passed on to the next handler in the chain.

When the handler task receives an exception message, it must either kill the offending task with a Destroy call, fix whatever went wrong and cause the task to resume execution by replying, or refer the exception on to the next exception handler in the chain. If the handler chooses to remedy the problem, the handler can manipulate the task's state (TTaskState) through the exception message's GetTaskState and SetTaskState calls and manipulate memory with Read and Write calls to the surrogate section (the entire address space of the task is available to the exception handler while they are in rendezvous). If the tasks are on the same team, the handler can also directly manipulate the

memory of the task that caused the exception. After the handler task remedies the problem it does a Reply, causing the offending task to resume execution.

For a hardware exception it is important that the handler task either remedy the problem that caused the exception, kill the offending task, or refer the exception on. Otherwise, the error will recur, creating an infinite loop. The onus is on the programmer to insure a correct resolution. If the handler task itself triggers an exception that it was registered to handle, both the handler and the original errant task are terminated.

Where possible, a machine independent class (THardwareException) wraps hardware exceptions. However, some hardware exceptions are highly machine dependent, so the task state and relevant exception data differs from machine to machine. For each machine configuration a definition file describes the machine specific classes for the machine state records, possible exception types, and exception message information.

Exception Handling Classes

Exception Message Hierarchy

TKernelMessage → TExceptionMessage

Exception Handler Hierarchy

MMessage-Dispatcher → TExceptionHandler

56 Elixir

(1) A substance held capable of prolonging life indefinitely.

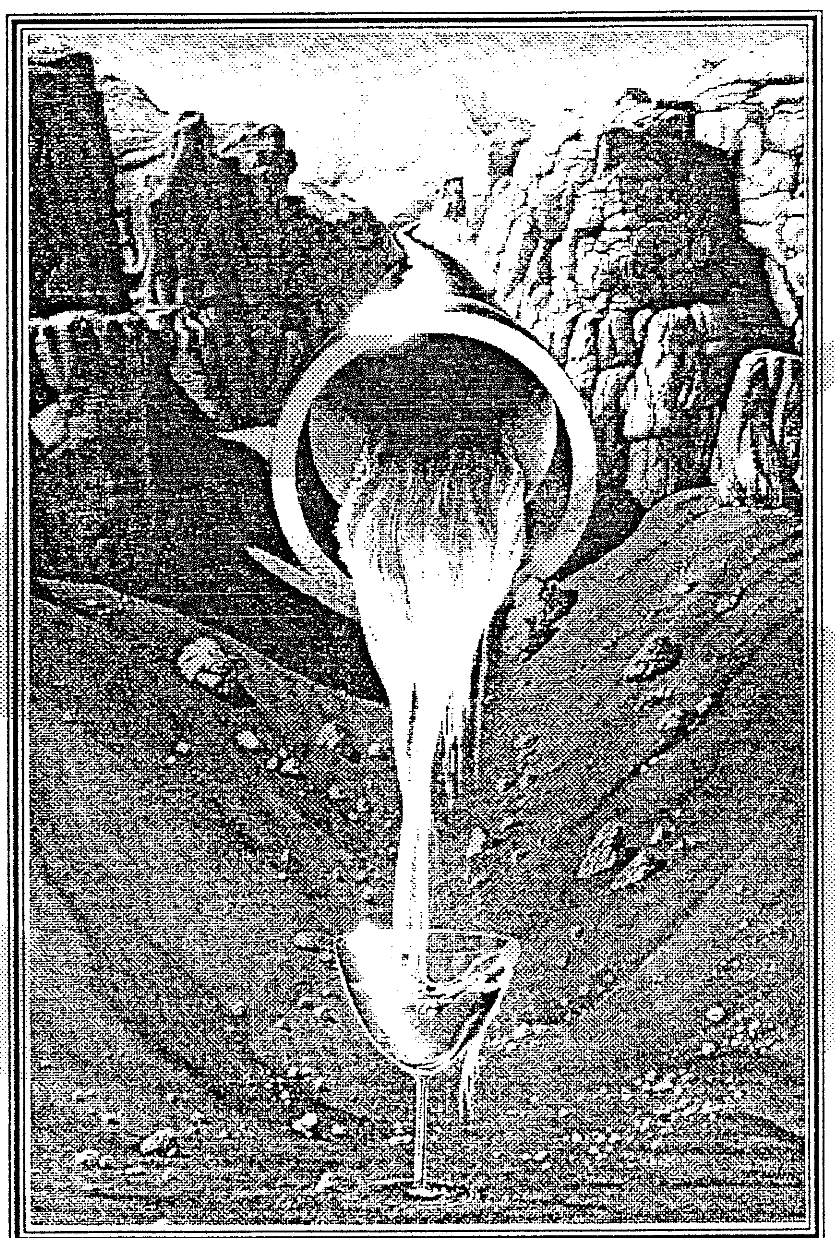
(2) Cure-All.

(3) A sweetened liquid usually containing alcohol that is used as a vehicle for medicinal agents.

(4) The essential principle.

(5) The Pink IO System.

56



56

Elixir IO System

Introduction

The Elixir IO System is a machine-independent IO model for the movement of data between the hardware attached to the system and the software within the system. While the Elixir IO System has the most machine-dependent code of the Pink system, it is based on an architecture the allows it to be ported to different hardware architectures as well as different configurations.

Architecture

The Elixir IO System has one primary goal: reduce the frustration of moving both hardware and software forward while not sacrificing the user's experience of the Macintosh. The user's experience is primarily based on ease of use, perceived performance, and compatibility. In the 90's, Apple must make major changes in both hardware and software, and we must be able to capitalize on these changes quickly. For Apple to be successful, our third party developers must be able to track these changes as easily as we do.

When the Macintosh was first shipped we only had third party software developers. Once we opened up the Macintosh, we found many third party hardware developers meeting user needs that Apple could not meet (big disk, large screens, etc). As our systems become more complex, our dependence on third party hardware developers will only increase. Clearly the magnitude of our success will be related to the success of our developers. Therefore our third party hardware developers must be able to reap all the same benefits that Apple gets with the Pink system.

The role IO plays in the overall system performance can't be understated. The relentless pursuit of IO performance has been the consummate force behind most traditional IO designs. The consequence of this pursuit has been that IO traditionally has been one of those forbidding and complex parts of the kernel where only gurus tread. The Elixir IO System is taming this complexity by providing services based on simplifying abstractions such as Interrupt Service Routines, Access Managers, and object oriented frameworks.

Booting the Pink system from multiple possible data sources is the other major design center for the Elixir IO System. These data sources will include ROM, hard disk, and the network, but will not be architecturally limited to them. The role of ROM for booting can vary greatly. At a minimum, the ROM would contain just enough code to boot using another data source. The other end of the spectrum would contain the entire Pink system in the ROM.

The primary goal of the Elixir IO System spawns a set of secondary goals. These are: reduced complexity and time to create a new device driver; automatic configuration of most, if not all devices; the reuse of most IO code whenever possible. Most IO policy issues regarding nearly all tradeoffs are architectural pushed down to the lowest level possible. Network services are integrated from the start as opposed to grafted on later.

The function of the IO driver has changed significantly with the advent of the Elixir IO System. The classical IO driver does very little functionally, but often must fight the system if it wants to do anything non-standard. IO policy issues are dictated to the classic IO driver by the rest of the IO system (request queuing, device allocation, access policy, etc). Most classic IO drivers are forced into some "standard interface" form like open, close,control, etc of the classic Macintosh. That was the old way.

The Elixir IO System tries to create "standard interfaces" only when and where they make sense. Standardized interfaces should only be used to give *good value* for their constraining abstraction. An example of a very useful standard interface is the one between the file system and the mass storage devices. By abstrating away all of the messy details, the file system may use very diverse devices like hard disk, CD ROMs, tapes, MO disks, floppies, etc.

Each type of IO device is likely to have differences in how it is to be accessed. Some devices can't be shared (printers) while others can (networks). Printers can give the illusion of being shared by *spooling* output to disk. Cards found on expansion buses may have many devices with different access policy issues. Clearly device access policy can't be correctly predicted for all devices today. Therefore the Elixir IO System tries **not** to make IO policy. Any device access policy that we would impose today would most likely be incorrect some day in the future. The Elixir IO System solves this issue by moving as much of the policy issues down to our new drivers. The functional role of our new drivers has been expanded from just the simple data movement and control of a device to also include the definition of access policy of the device.

Interrupt Service Routines and Access Managers

At the lowest level, the new Pink drivers are made up of an interrupt component and a task. The interrupt component runs within the kernel's address space and is called an *Interrupt Service Routine* (ISR). The task part of the driver handles all non-interrupt aspects while executing in an address space outside the kernel. Since the task also defines the access policy of the device, we have named the task component of the new drivers, *Access Managers*.

Under the Elixir IO System, what implementation details that are done in the ISR or the Access Manager are not cast in concrete. However, given proper hardware support and the antisocial behavior of long interrupt processing, most ISRs should be small. If there is insufficient hardware support, then the ISR may be as large as necessary. Because the Elixir IO System is based on a true multitasking system, we will encourage more social IO hardware.

Interrupt Manager

The Interrupt Manager embodies a set of interrupt services found in the Opus kernel. The Interrupt Manager has two major purposes: the timely dispatch of interrupts to the correct ISR and the migration of ISRs within the kernel. Access Manager installs and removes ISRs in the system by making requests to the Interrupt Manager.

Configuration Access Managers

IO devices can be attached to a system via many diverse hardware paths. Some are built in on the motherboard, some are attached to buses (ADB, NuBus, SCSI, BLT), while others are a mixture of both, a NuBus card with a SCSI chip on it. A simplifying abstraction is to view these different hardware configurations as a hardware hierarchy. Viewing the hardware as a hierarchy infers we should naturally view the *software* for these devices as a hierarchy. The hierarchical view of software fits nicely in restricting the scope of knowledge to obvious layers of the hierarchy. By limiting the scope of knowledge we can easily push IO policy issues to the lowest levels of the hierarchy.

Configuration Access Managers are responsible for a collection of devices. The obvious example of this type of access manager would be the ADB access manager and its interrupt code. Besides providing all of the simplifying abstractions of the bus (interrupt decode, read access, etc), the ADB access manager will also be charged with the configuration of the devices on the ADB bus. When any *bus* access manager is started up, it would have the responsibility to find all the devices on the bus. After the devices have been found and identified, the given bus access manager would make the policy decision to spawn the appropriate access manager or just record that the device was found but not linked with an access manager.

The second type of configuration access manager would be found on expansion cards (NuBus, BLT, etc). This type of access manager would most likely know exactly what devices are on its card, the exception being the case of another bus on its expansion card. This model of configuration access managers can be applied recursively for the bus found on an expansion card. The use of software hierarchy to manage an arbitrary hardware hierarchy allows the Elixir IO System to configure any hardware platform or configuration.

Frameworks

A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes. The reuse of most of the non-interrupt IO code will be the cornerstone to meeting our primary goal of reduced frustration in IO development. The framework will embody all of the complex system interface issues. In this way, the level of effort to add a slightly different device to an existing framework will be inconsequential.

It is our stated intention that third party developers will be able to capitalize on our frameworks to the same extent as we do. In this way our new systems will quickly have a diverse set of devices with functionally correct interfaces.

The Elixir IO System will group only those parts of the problem that are related into one framework. At this time, we will not attempt to have one all-encompassing IO framework. This may change as we all become more experienced with frameworks. The current frameworks we are considering are, Mass Storage, NuBus/Video, and ADB.

56

Pink Booting Overview

Introduction

Designing the boot of the Pink system is one of those good news, bad news, stories. The good news is we get to use object oriented frameworks to solve a really messy set of problems. The bad news is a question: what comes first, the boot of the system or support for object oriented frameworks? The use of object oriented frameworks has created many circular dependencies with regard to booting.

A comprehensive design of booting will take several months to complete. This document will give a simple overview of how we plan to boot the Pink system. A key part of this overview is a list of design goals and open issues.

Clearly our design must solve the object oriented circular dependencies. With some very helpful input from Andy Heninger, the design outlined below should address most of these issues. The key to this solution of circular dependencies is to create a memory image with a collection of severs bound to a set of shared libraries. More about this solution latter.

Booting Goals

The single biggest bottleneck Apple has shipping a new CPU is the amount of time it takes to create and test the system software. Much of this effort will be simplified with our overall object oriented design. The boot process design is an opportunity to further reduce our time to market without unduly constraining the hardware designs. Therefore the portability of booting code will be a major design goal.

The role of ROMs in our current Macintosh are well known to all of us. Due to performance reasons, some hardware designs centers don't want to require large ROMs. Our boot design will, at a minimum, use ROMs to read in the rest of the boot data from some other data source. Lower cost design centers would like to reduce cost by trading ROM for RAM. Our design will accept this design decision as well.

Booting from a Blue-ROM-based Macintosh is also a design goal. If the system were to be converted to a Pink-only system, we would come in at what is known as *Boot Block Time*. If the system were to support both Pink and Blue then some form of application would be used to load Pink, as we do today.

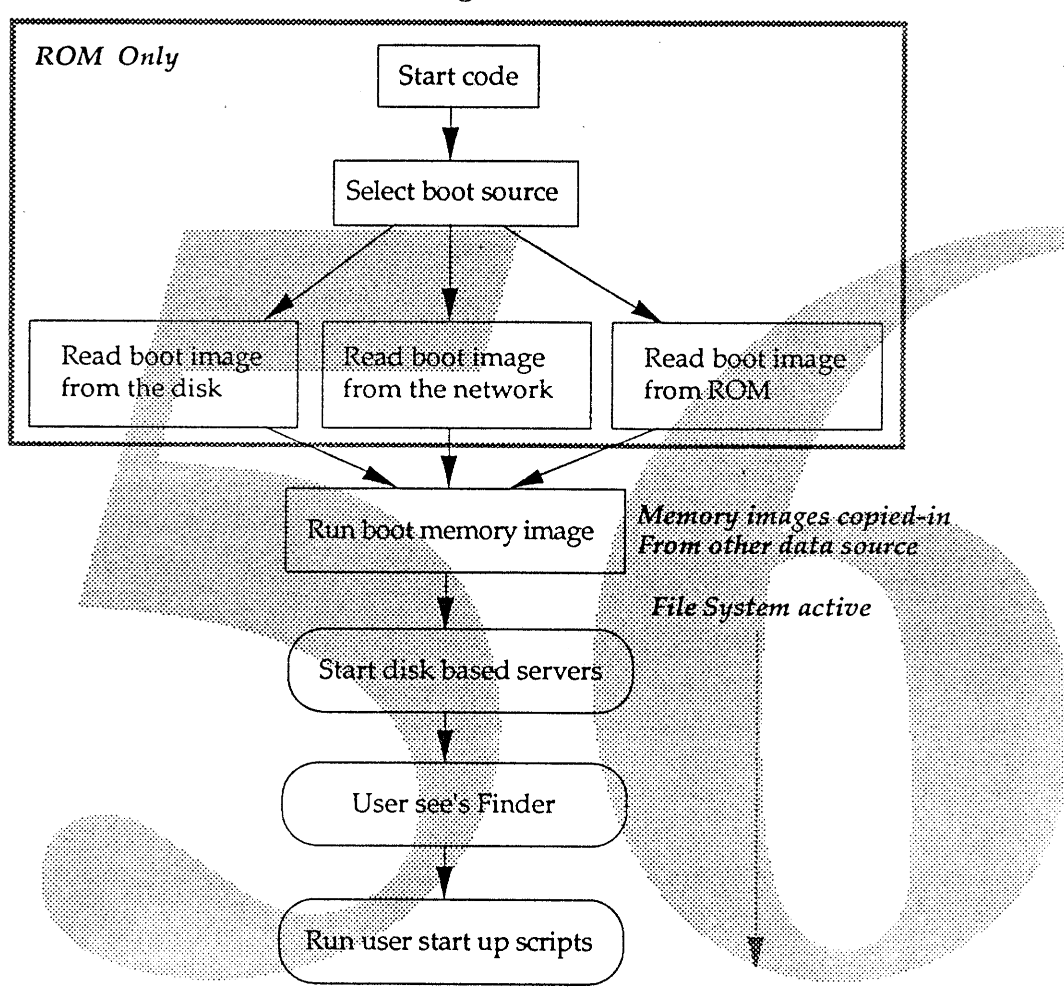
The use of a local area network as a boot device makes a great deal of sense in many of our markets. It should be much easier to achieve this goal if we integrate it into the booting design right from the start. Therefore, booting from multiple data sources will be a design goal.

Third party developers will need to join the booting fun, but not at the expense of the user's ease of use. This is a hard issue because of the tradeoffs. If we constrain our developers too much, we lose interesting applications. If we open it up too much, the user may get confused. Simplicity and expansion with third party access will be another design goal.

The Pink Booting Sequence

The figure below gives a simple block diagram of how the Pink system will be booted. There are three major sources of execution: the ROM *Start Code*, the *Boot Memory Image*, and the servers found in the Pink file system.

Pink Booting Overview



The Start Code

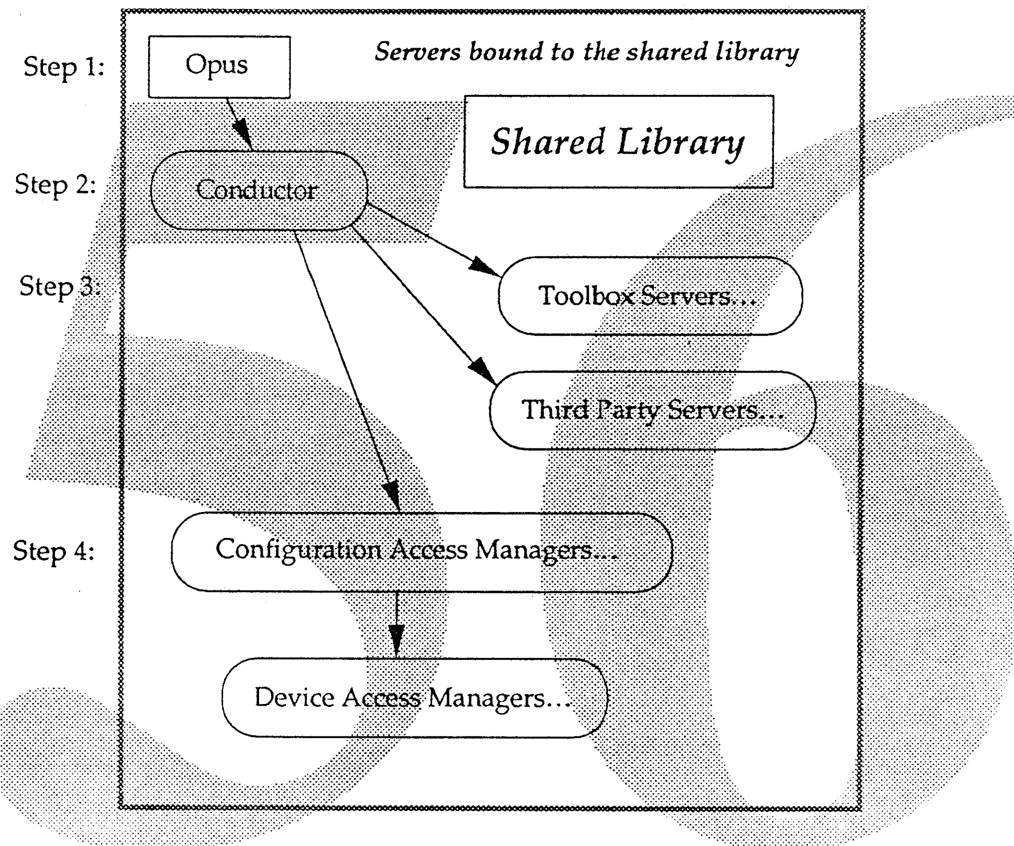
The Pink start code design is clearly a descendant of the Blue start code design. Our start code is austere compared to the Blue code. The universal ROM idea for a family of design centers is also embraced by the Pink code. Our simplified start code will contain:

- Memory configuration information.
- Diagnostics for memory, IO chips, etc.
- System feature configuration.
- Default configuration of some motherboard hardware (VIAs, etc).

Boot Source Selection

The details on how to select a boot source has yet to be worked out. The selection criteria will be based on IO hardware configuration and some sort of state information (PRAM?). Example, in the current Macintosh booting from a floppy has presidents over booting from a hard disk. Once the selection process is complete the ROM based IO code selected will read in the boot memory image.

The Boot Memory Image



The Boot Memory Image is the key to resolving the object oriented circular dependencies. The boot image contains: the Opus kernel, the Pink file system, the *Progenitor Set* of toolbox servers bound to a shared library, and a collection of memory resident files that the file system has access to.

The Progenitor Set of toolbox servers are linked with their shared libraries to create a single memory image. This memory image contains enough toolbox servers such that all circular dependencies have been resolved. The complete set of toolbox servers required is not known at this time. This same image could be mapped into many address spaces. By selecting the correct starting PC, that address space becomes the given toolbox server. (Neat trick. You can thank *Andy Share My Library Heninger* for it)!

Another new feature of the boot image involves the file system. This trick allows third parties to add to the boot image, but not be statically linked to our shared library. The Elixir IO System also plans on using this feature. The details on how this works is still not defined, but roughly it goes as follows. A magic attribute (folder?) is defined to be the boot image attribute. When a new file is added to this attribute set, the boot image is automatically updated to contain this new file. Once the file system is started up, but before the backing store servers have been started, these boot files are memory resident. Using standard file system access methods, any server in the Progenitor Set can access these files via the file system. A simple idea, but somewhat harder to implement.

The last new feature is called the *Conductor*. The Conductor is the task who's job it is to sequence through all the remaining parts of the boot process. How the Conductor knows the sequence is still undefined at this time.

Now that we have defined all of the services, lets see how it works.

Step 1: The Opus kernel is started and the file system.

Step 2: The Opus kernel starts the Conductor.

Step 3: Each of the Progenitor Set servers are started up.
Any third party servers are started up.

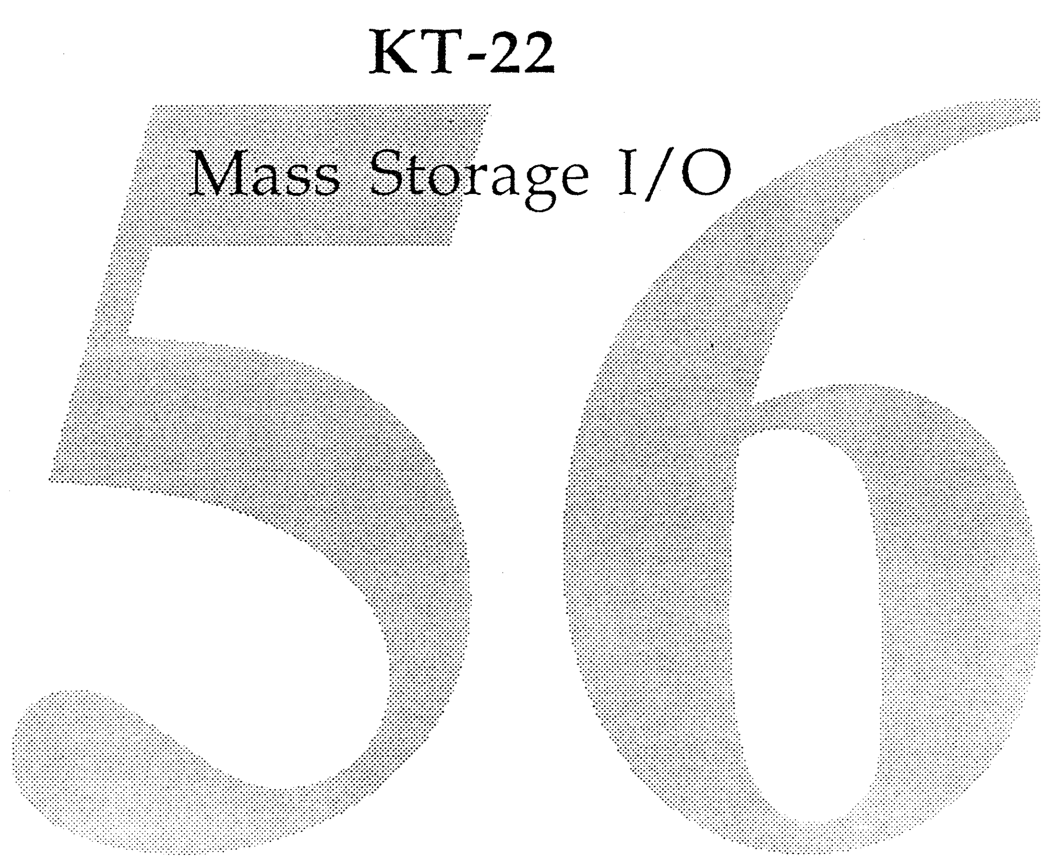
Step 4: Each of the configuration access managers are started up.
These in turn will start up the necessary device access managers.

At the end of Step 4 the backing store is completely up and all other services can now be loaded from there. Because the Elixir IO System is held off until the Progenitor Set has run, the IO system can use all the same object oriented services used by the toolbox.

Many details still have to be worked out, but this overview gives us a target to work against. It should be noted that the boot image will differ for a disk-based boot as opposed to a network-based boot. Still, the outline is the same.

KT-22

Mass Storage I/O

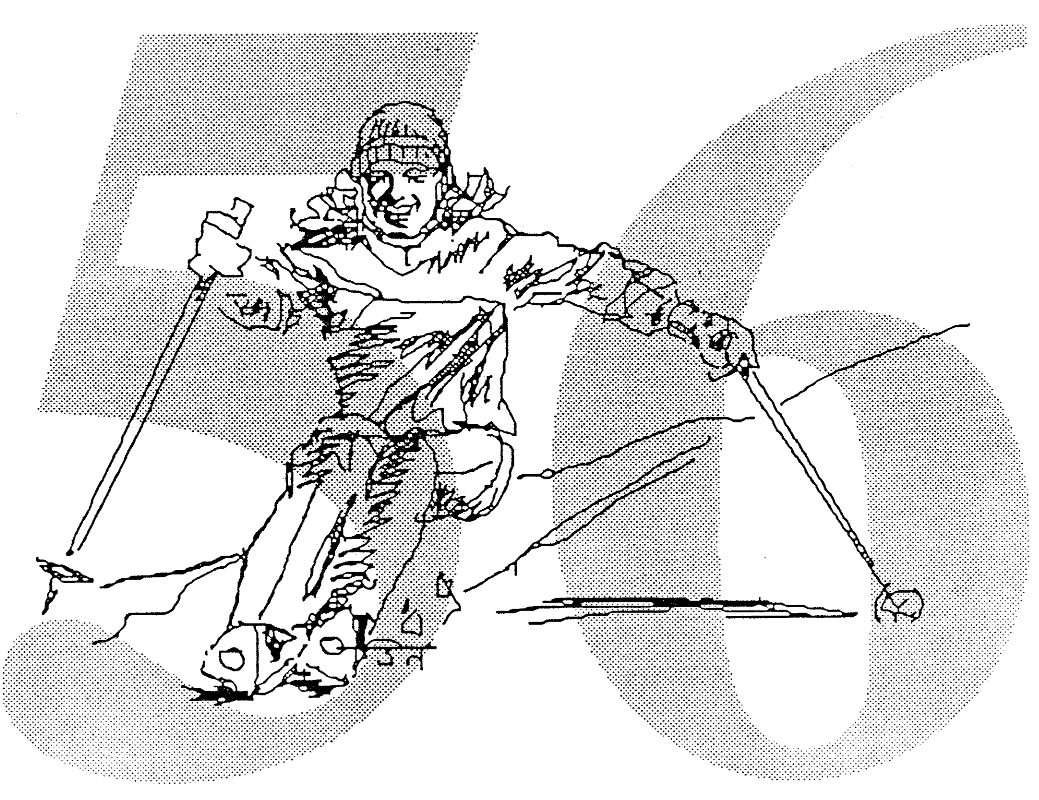


56

KT-22

Mass Storage I/O

Bob Otis
x4483



Skiing KT-22 is like I/O - everybody wants to get through it quickly regardless of the obstacles placed in front of them. Sometimes they are both painful experiences.

56

Overview

The purpose of this document is to set a direction for Pink's Mass Storage I/O Model. My definition of Mass Storage is; all devices that are used for permanent storage of data. Some examples would be hard disk drives, floppy drives, and Magneto Optical drives.

Pink's Mass Storage I/O model interacts with Pluto (for file I/O), Opus/2 (for page faults) and any other task that needs information from this type of device. The key components of this model are Device Access Managers and Bus Access Managers. The roles of each of these areas is defined below.

The following topics will be discussed in this document:

- Major System Components
- "Volume" vs physical devices
- Device Access Managers
- Bus Access Managers
- Issues

Major System Components

The Mass Storage I/O model is a major corner stone of the Pink System. With a virtual memory system, more of a burden is put on the mass storage devices, as pages of memory are continually swapped in and out of the system. The goal is to accomplish this with as little system impact as possible.

Below is a summary of the three components of the I/O model and their responsibilities; the Device Access Manager, the Bus Access Manager and the Interrupt Service Routines (ISR). As shown in the diagram below, both the Device Access Managers and the Bus Access Managers are OPUS/2 tasks in user space.

The responsibilities of the Device Access Managers are the following:

- Interface with the Pluto.
- Interface with the kernel for paging I/O.
- Handle device specific information.
- Interface with the Bus Access Managers.
- Interface with the DeskTop Objects.

The responsibilities of the Bus Access Managers are the following:

- Interface with the Device Access Managers.
- Handle hardware specific characteristics.
- Handle device interface characteristics, such as SCSI protocol.
- Manage different sources of hardware; NuBus cards and Direct Slot Cards.

The responsibilities of the Interrupt service routines are as follows:

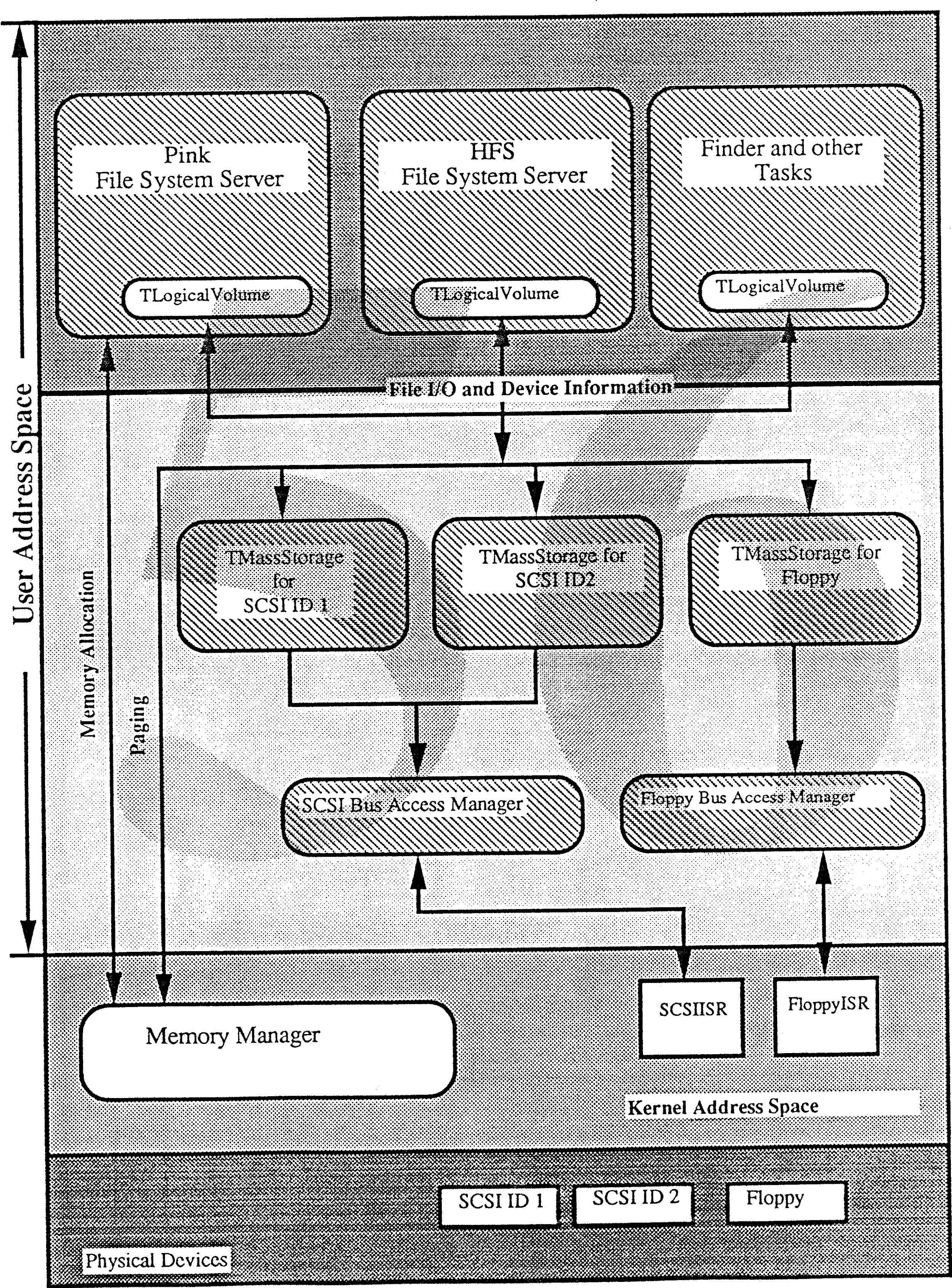
- Handle hardware interrupts
- Perform any I/O required.

More detail can be found for each of the areas throughout this document.

Mass Storage has at least three clients all with different needs. The three clients I will describe here are the kernel (Memory manager), Pluto and the Finder.

The kernel needs the I/O model to handle page faults as fast as possible. The interface to the kernel will be a frequently used for swapping memory in and out of the system. Pluto's interface will be used for standard file I/O and all other normal interface to mass storage devices; Format,Verify, Eject and others.

Mass Storage I/O

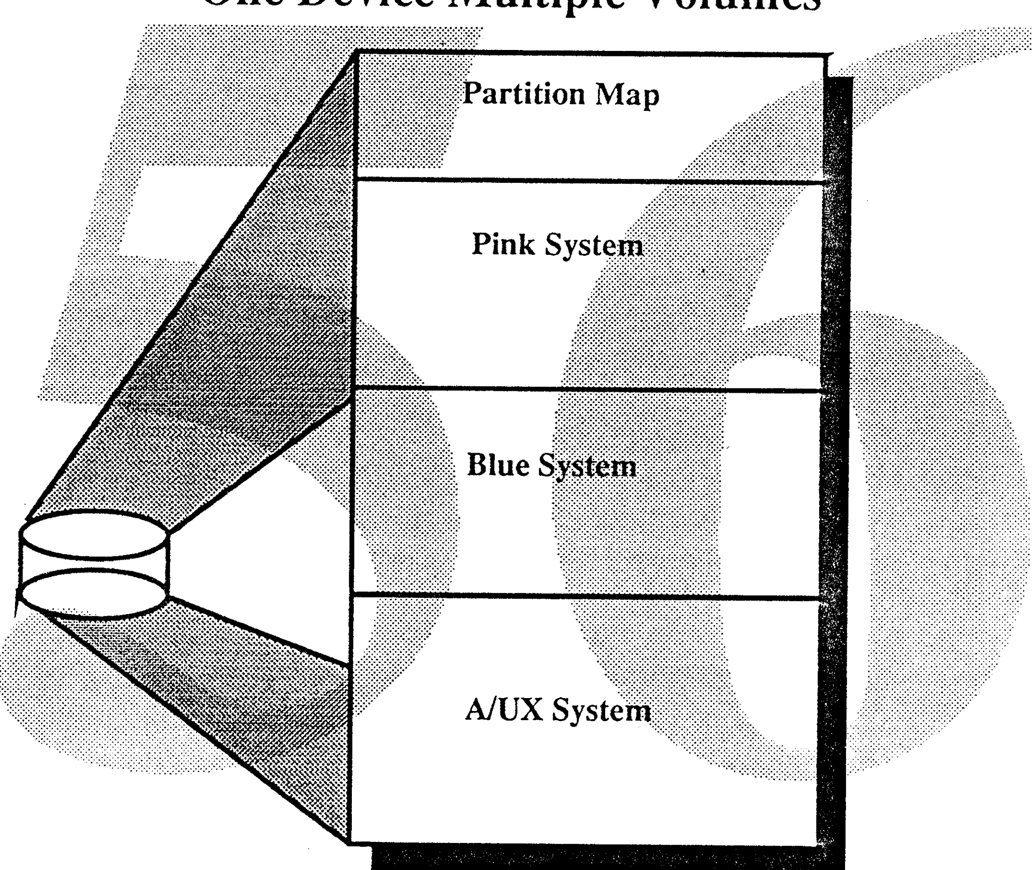


Volume vs Physical Device

To understand the Pink Mass Storage model the relationship between the "Volume" and the physical device must be explained. There is not a one to one mapping between the "Volume" and the physical devices. Normally, many "Volumes" will be on one physical device as shown below.

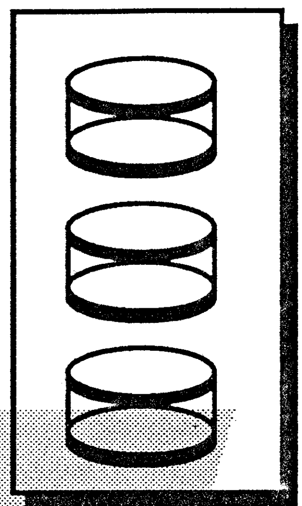
In the Macintosh world today SCSI devices are divided into different partitions for different operating systems. To the different file systems involved this looks like multiple "Volumes", but to the Device Access Manager, this is one physical device. It is beyond the scope of this document to describe the partitioning in detail.

One Device Multiple Volumes



The other relationship between "Volume" and physical devices is one "Volume" that maps to multiple devices. As shown below, this relationship between volume and devices is hidden from the Macintosh file system of today. If this relationship was available to the file system or a data base system the devices could be used more efficiently by evenly distributing the data across the different devices. One goal of the Device Access Manager framework is to make this information available to it's clients

One Volume Multiple devices



Device Access Managers (TMassStorage)

Overview

The mass storage framework is to device access manager writers as the application framework is to application writers. The mass storage framework defines the file system and kernel interface to all Mass Storage Devices. The framework allows a new Device Access Manager to be written without having to be concerned with the system interface specifics. The Mass Storage Framework follows the Pink philosophy of making the developers life easier.

Adding mass storage devices to systems in the past has been non-trivial. The driver writer had to understand the following; file system interface, hardware or "Manager Interface" and the specific device characteristics. This burden made it difficult to add new devices quickly.

Mass Storage driver writers today have it easier then most of the other driver writers in the current Macintosh system. The most difficult thing to understand is exactly what the 5 calls; Open, Close, Control, Status and Prime are supposed to accomplish. Once they understand the 5 calls they must understand how to interface with the manager (SCSI Manager) or the hardware itself (floppy drivers).

The SCSI Manager has taken the first step by hiding the hardware implementation from the SCSI driver writers. This allows hardware changes to the SCSI port without modification to the driver. However, the SCSI driver writers still have to worry about what calls to make to the SCSI Manager. With the advantages of the object oriented paradigm, the file system interface to the Device Access Manager and the underlying hardware manager interface will be hidden, unless the Device Access Manager has a great need to add to the file system interface.

The Device Access Manager writers (Pink's equivalent to driver writers) will only have to concern themselves with device specific software. For example: The SCSI Device Access Manager Framework will contain member functions that allow all of the common SCSI commands to be sent to the device. If a new command or modification of an existing command is needed, the necessary member functions will need to be added or overriden.

Goals

When a project begins it is important to define the goals and objectives. This section describes the goals and objectives of the Mass Storage Framework. Below is a summary of the goals followed by a more detailed description of each one:

- To make writing mass storage device access managers easy
- To use the Object-Oriented paradigm
- To use Pink Toolbox wherever possible
- To be Fast and Efficient
- To hide device specific characteristics
- To aid (not add) to the real time characteristics of Pink
- To support all of Pluto's needs
- To allow a fast I/O path for page faults from the kernel
- To support all of Finders needs
- To support new device types for the next 5 to 10 years

Pink's overall goal is to make developers life easier. The mass storage framework adheres that goal, making writing of device access managers easy. The class TMassStorage has member functions that handle all of the file system and kernel interfaces. An engineer adding a new device to existing I/O channel will not have to worry about the details of the File System interface. Different classes will be defined to allow easy interface to the bus access managers of a specific I/O channel. For example, MSCSIDevice is a class that allows the device access managers to interface with the SCSI Bus Access Manager.

The mass storage framework will be written using OOP. This allows the framework to be shared between many different device types; SCSI, Floppy or any new device types added in the future.

Whenever possible the framework will take advantage of the Pink ToolBox. The Opus/2 wrappers and Utility Classes are two examples. However, everything that is used by the device access managers must be available early on in the boot process, as well as locked down in memory. WE CANNOT PAGE FAULT.

Everyone associates slowness with C++. The mass storage I/O path must be fast. IT WILL NOT BE SLOW.

The purpose of Device Access Managers are to manage device specific characteristics, allowing the file system and kernel to be device independent. All non essential device information will be controlled by the device access manager. Some device specific information will be available to allow better utilization of the device.

Pink has been defined as a real time system. The Mass Storage Framework is a key part of making this work. The framework will export as many functions as possible to allow real time access to mass storage devices.

One client of any device driver is the file system. The Mass Storage Framework will export all services that Pluto requires.

Another client is the Memory Manager to service page faults. This requires that a special I/O path is made available to the kernel.

The Finder group has indicated that they will model the real world on the desktop. In order to accomplish this, services will be exported from the framework that allow some measurement of activity to be displayed. The resolution of that activity is still to be defined.

Since Pink is to be Apple's OS of the 90's, the Mass Storage Framework must prepare for the changes in mass storage devices. This will require support of larger and faster devices. Today 80 megabytes is a standard size on Apple's mid-range computers. By 1995 it's anybody's guess what will be the standard. Today Magneto Optical devices are available with up to 150 megabytes per side as well as a 1 gigabyte 5 1/4" full high disk drives.

Architecture

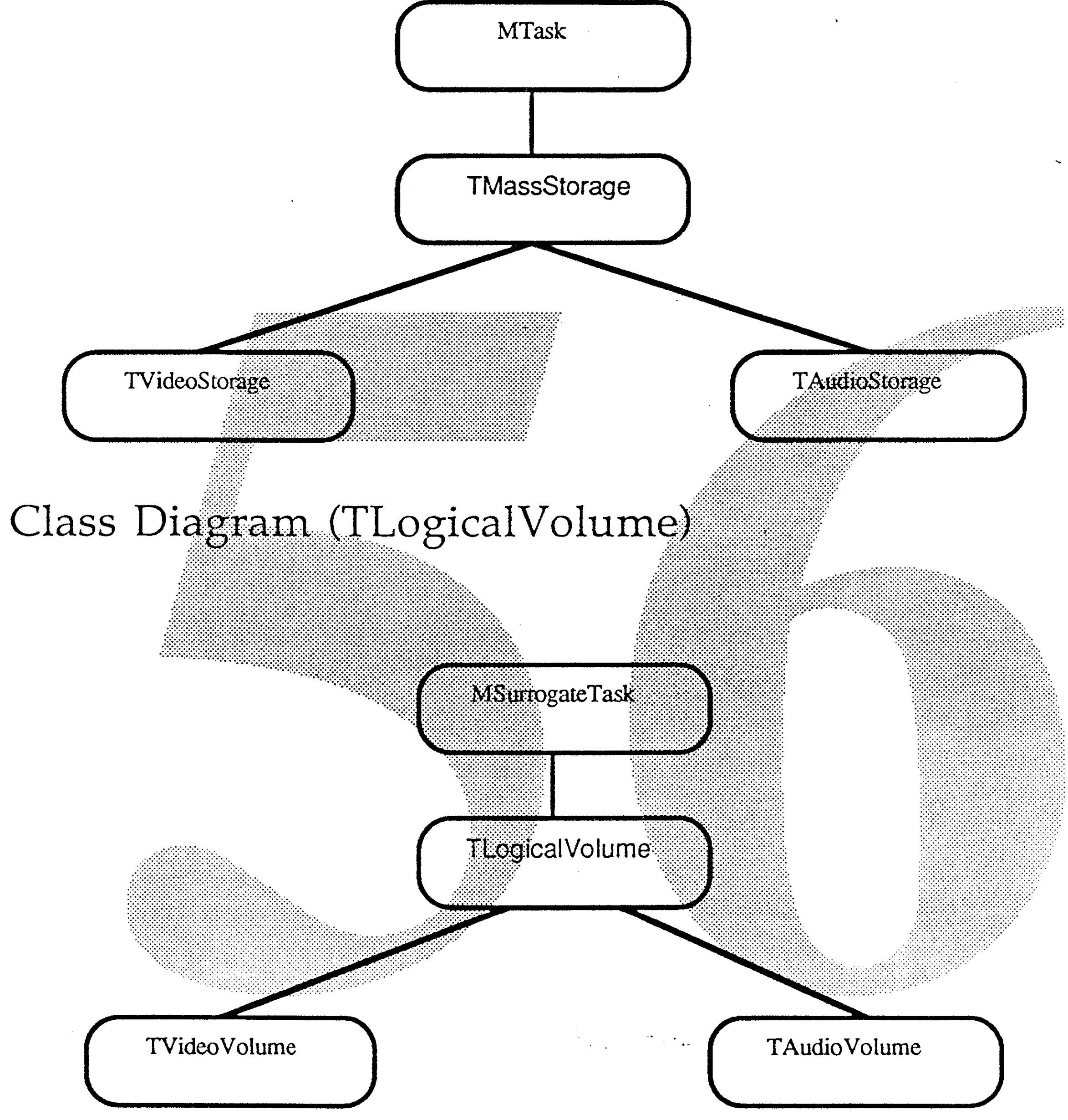
This section describes a set of Base classes that are subclasses of MTask. These classes can be combined with a device interface class, such as MSCSIDevice to create a mass storage device access manager. The purpose of these base classes is to give a framework for all Mass Storage device access managers. There are two classes defined here, the client (TLogicalVolume and it's subclasses) and the server (TMassStorage and it's subclasses).

TMassStorage is an abstract base class designed to allow quick development of mass storage device access managers. This class must be subclassed and combined in a **has-a** relationship with an interface class such as MSCSIDevice to create a mass storage device access manager. The device access manager tasks created by using TMassStorage handles all of the normal driver functions; **Reading, Writing, Formatting, Ejecting of media** along with other needed functions of any mass storage device.

A set of base classes are defined for other mass storage device types. This allows a framework for devices such as CD-ROM (TAudioStorage) with an audio interface to have a standard set of interface calls such as; **ReadTableOfContents, WriteTableOfContents, PlaySong, SearchForSong** and others.

The client class (TLogicalVolume) is instantiated by a task that needs to access the volume supported by a TMassStorage device. A list of tasks that would instantiate this class would be the file system, kernel (memory manager) and the DeskTop Manager. To the file system and kernel this path would be used to access a specific volume. The DeskTop Manager (interface still to be determined) would need device level information.

Class Diagram (TMassStorage)



Class Diagram (TLogicalVolume)

Mass Storage Bus Access Managers

Overview

The Mass Storage Bus Access Managers are policy makers for a specific mass storage bus type. Mass Storage Bus Access Manager is an OPUS/2 team that is responsible for determining the boot policy as well as the access policy for device access managers. Each bus access managers is responsible for managing access to a hardware chip and the mass storage devices attached to it.

For example, the SCSI Bus Access Manager is responsible for the NCR5380 SCSI interface chip and all of the SCSI devices attached to the SCSI Bus.

Each bus access manager consists of two functional areas; The configuration manager and the access manager. Below are the responsibilities of the configuration manager:

- Launching the Access Manager
- Scanning the appropriate bus for attached devices
- Identifying the device types, sizes
- Locating and launching the associated device access manager (If available)

The responsibilities of the access manager are as follows:

- Manage all access to a specific hardware on the mother board
- Hide the hardware implementation from the Device Access Managers
- Understanding the specific protocol of the bus

Issues

Below is a summary of issues that will effect the framework design:

- Dependencies on Wrappers and Runtime may cause boottime problems.
- Everything the framework depends on must be locked down at all times.
- Desktop Service; What are they?
- Things to do for Mass Storage devices

  - Format utility
  - Ability to form a volume out of multiple devices.
  - Backup utility

- Dynamic characteristics of the media needs to be handled. How does the Desktop Manager and file system want to accomplish removable media?

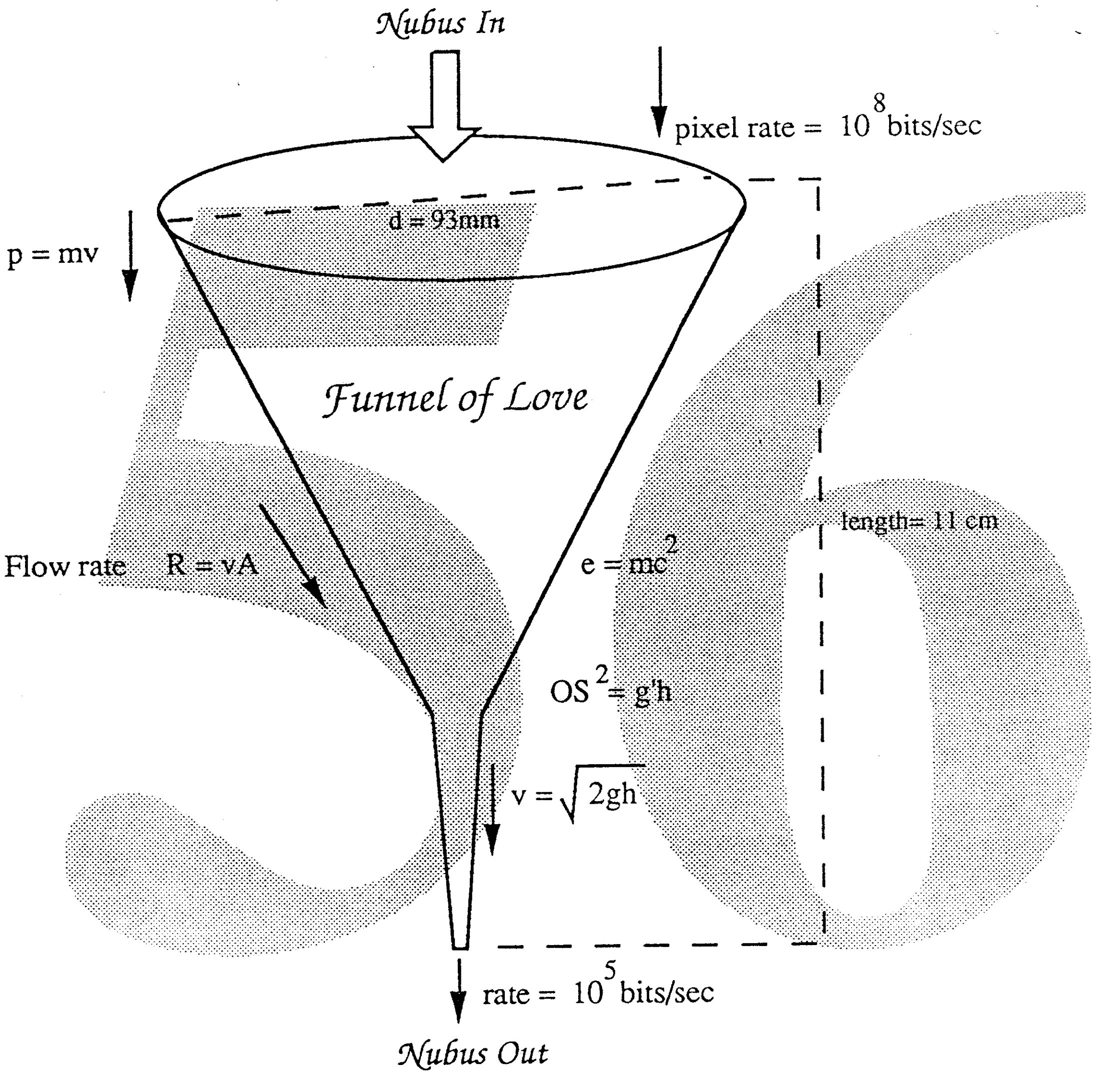
56

Funnel of Love

56

Nubus I/O Framework

56



Nubus In
pixel rate = 10^8 bits/sec
d = 93mm
p = mv
Funnel of Love
length= 11 cm
Flow rate R = vA
e = mc^2
OS^2 = g'h
v = $\sqrt{2gh}$
rate = 10^5 bits/sec
Nubus Out

56

Architecture

Design Philosophy

The basic philosophy of the Nubus I/O Framework design is this: make Nubus easy for the programmer to understand and use. Currently, using the Blue Slot Manager, the programmer will not get very far without understanding an entity called the *sResource* (short for Slot Resource). Also fundamental to the Blue Slot Manager are the Slot Parameter block and the SExec Block - both being large structures filled with fields such as *seFiller1, seFiller2, spMisc,* and *spKey.*

The "naive" programmer may wonder what in the world all of this has to do with Nubus, the Nubus cards that he is interested in, and the Nubus card's Rom and Ram. This "naive" user is absolutely correct; the programmer needs a view of the Nubus space that is more intuitively usable. Simply put, the easier to use, the better.

For this reason, the fundamental objects in the Pink Nubus I/O Framework are: Nubus itself, the Nubus cards, and the Slot Resources. The Slot Resources follow the standard Blue Slot Resource format in order to provide compatibility. However, the programmer must neither understand large parameter blocks nor view the entire Nubus system from the standpoint of Slot Resource structures.

Class Overview

The Nubus Framework allows programmers to access the slot devices resident on Nubus. The Framework provides a base class that allows clients of the Nubus Framework to obtain information concerning the Nubus cards that are found in the slots. The Nubus Framework also provides a base class that represents an individual card in a Nubus slot. This class provides methods for accessing the actual slot Resources, and other structures that reside on a given card in a Nubus slot.

For the sake of future flexibility, we will be interested in an abstraction of Nubus that is more general: we'll refer to this as (TExpansionBus). This will be a superclass of the Nubus base class, a virtual base class, that can be overridden for other system objects that are not Nubus but are slot expansion busses. Examples of this would be a (TBusLikeThing) and a (TMACseDirectSlot).

Client Overview

An example of a client that uses the Nubus and Nubus card classes to access card information is the (Nubus Access manager). This access manager, a team, walks the Nubus slots to determine which cards are resident in the system.

A client that is only interested in the slot Resources that are on Nubus, and not the particular physical layout of Nubus cards, may want to simply use the Nubus base class to provide access to the collection of total slot Resources that are on all of the cards.

The Physical Nubus World

In a sense, the Nubus I/O Framework is meant to model the physical world of Nubus. In some cases, the real world is so terribly abhorrent that we must attempt to shield the user from the details of Nubus's wretched persona. In other cases, the "real" or physical world of Nubus is a perfect model for the software interfaces. One of the design goals of the Nubus I/O Framework is to model the physical in the cases where such as model will be intuitively easy to use, and to provide higher levels of abstraction where the physical world lacks clarity.

For example, our machines have Nubus slots, and Nubus cards go in those slots. Our computers may have anywhere from zero to sixteen Nubus slots (the zero case not being especially interesting). The class definitions for the Nubus Framework will allow users to address the Nubus as a whole as well as the many cards that may be installed on the bus. The classes will be flexible enough to handle any possible configuration of Nubus cards. The basics of communication with each card are a well defined standard, and will allow the Framework to provide at least a minimum amount of communication with every card.

Some machines, such as the SE30, have pseudo-Nubus slots. These pseudo-slots will be handled just as if they were actual Nubus slots on a Macintosh II style machine.

Each Nubus card has a declaration ROM in its address space. The clients of the Nubus Framework will use the class member functions to access the declaration ROM on each Nubus card and thereby identify the card's special capabilities. Predefined, declaration ROM data structures called slot Resources are used to initialize and configure a Nubus card.

The applications and most of the clients of the Nubus Framework will be insulated from the low-level operations on slot Resources. The basic functionality that must be accomplished can be summarized as follows:

- Location of Nubus cards with declaration ROMs
- Loading card's device drivers from their declaration ROMs
- Running the declaration ROM initialization code
- Initializing and using the parameter RAM on the host system during system startup
- Accessing the information on the card that resides in different slot Resources
- Shielding the user from the frightening concepts of Bytelanes, Bus errors, and the various sResource types (if you don't know what these are, don't ask).

Clients of the Nubus Framework

The prospective clients of the Nubus Framework are mainly system software components. Applications will most likely use device drivers that are loaded of the Nubus cards, for the most part. The software components most likely to use the Nubus I/O Framework are:

- The boot system code,
- device drivers, and
- low level system utilities.

The boot code, for example, will need to walk the slots on Nubus and decide which cards he needs access to immediately. In the case of a typical local boot, the video cards may have their drivers loaded so that a happy Mac (hopefully) may be drawn to the screen that may be have been any pixel size, bit depth, and color mapping. The drivers themselves will then need to get more information that is stored in the declaration Rom slot Resources on its own card. The Nubus Framework will allow the card to access its own structures and then configure itself.

In the case of a remote boot, the device drivers for a communications card that will provide network access to a bootable stored image may be loaded. The drivers will then get the information that is stored in the declaration Rom slot Resources on its own card. The Nubus Framework will allow the card to access these structures and use the information to configure itself.

A system utility such as a "ResEdit" for Slot Resources Utility - let's call it the sResource_Browser - may need to reach down and grab device drivers and slot resource tables from Nubus cards. Admittedly, this would be an unusual and special utility. The Pink system will be dynamic and should support many types of reconfiguration that don't necessarily happen at startup time. Any type of reconfiguration or status reports that cannot be done through the currently loaded slot device driver will lead to the use of the Nubus Framework in order to directly access the card and its slot Resources.

Class Definitions

Class definitions and even DIAGRAMS! will be supplied in the near future.

56

Rob Lowe

Video Framework

56

Rob Lowe

Video Framework

Video star extraordinaire, Rob Lowe, is the mascot for the Pink Video Framework. Rob, the ultra-talented star of many a megabuck Hollywood production - as well as his own, highly rated, home-made videos - breaths new life into the old shop-worn term: Video Framework.

March 22, 1990
Jeff Zias
x44131

56

Architecture

Philosophy of the Design

The general philosophy of the Video driver design is the following:

The Pink Video drivers will provide an architecturally sound framework for the processing of specialized video calls to control the display of colors, animation, and other user modifiable video parameters.

The current Blue video device drivers provide the standard Open, Close, Status, and Control call interfaces. All of the different active video configuration calls are packed into the Control interface. The Pink video drivers will not lump the active calls into one large call.
The standard video operation will be something like this:

```
TVideo          myVideoCard;
myVideoCard.SetBitDepth(kOneBit); // Set your video card to 1-bit mode
```

The user of the Video driver will instantiate or access an already existing object of the proper video card type. Then, the member functions of the video object will provide the proper functionality.

Architectural Overview

The Video Framework will provide an abstract base class for the video system objects. Subclasses of the Video Framework base class will provide an interface for more specific video devices. An application may instantiate a class that has the ability to communicate with their specific video device, and then have the ability to control video device display parameters and color values. For example, the SetNewColors member function may be used for fast color cycling during an animation task's execution.

Applications such as color pickers, color matchers, and monitor configuration utilities may instantiate a Video object of a certain class in order to set color values, modify bit depth, or obtain the status of a given monitor.

The Video Framework class and its subclasses provide a higher level interface to the video device drivers - Access Managers and ISRs - that actually access the hardware.

Eventually, much of the functionality at the interface level of the Video Framework will be incorporated into the Albert graphical interfaces. For instance, a Pink Palette Manager may be the higher level interface for SetNewColors (color table modification) type calls.

System Requirements

The Pink Video drivers will be required to provide the following features:

- Ability to easily develop new objects or Access Managers for new Video hardware.
- Architectural foundation for implementing new animation methods.
- Object oriented framework that shields the user (the application and device driver programmer's) from needing to have any intimate knowledge of the video card's internal design.
- Reasonably fast, high-performance system.
- Hardware independent architecture.

Clients of the Video Framework

The components of the Pink system that will most likely be clients of the Video Framework are:

- the Palette Manager,
- Color pickers,
- Animation Manager,
- Cursor code,
- Applications that Color Cycle,
- the Boot code, and
- Utilities that configure video.

Major System Components

The primary components of the Video drivers are:

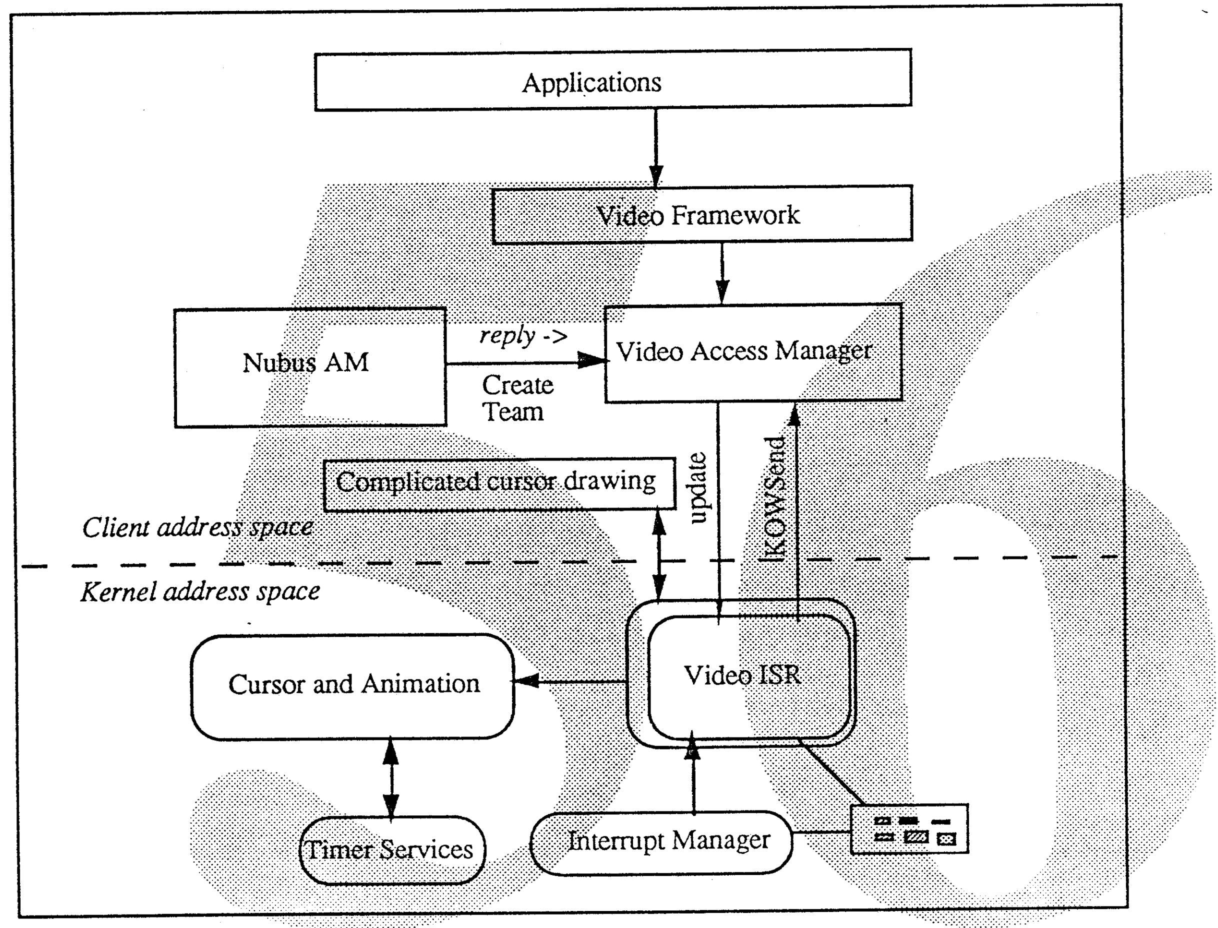
- the Video Driver Framework,
- the Video Access Manager, and
- the Video ISR.

The Video Driver Framework is the header files, class descriptions, and general application level interface to Video driver control. The Video framework will provide a generalized interface that may be subclassed in order to provide more device specific interfaces.

Information Flow of the Video Framework.

Pictured below is a proposed model for the implementation of the Video framework from a system component point of view. Keeping in mind that each task switch and each message can be fairly expensive, one of the primary design goals, speed, can possibly be increased by minimizing the number of IPC messages sent.

The cursor code is a special client of the Video Framework. The cursor code, for performance reasons, may need to be closely coupled with the Video Framework's internal design. As seen in the diagram below, the cursor code may need to run off the video interrupts and be ready to draw the cursor from either the system kernel level or from a client task's address space. For more information on cursors, see Don Marsh's cursor design document.



56

56 The Plumbing

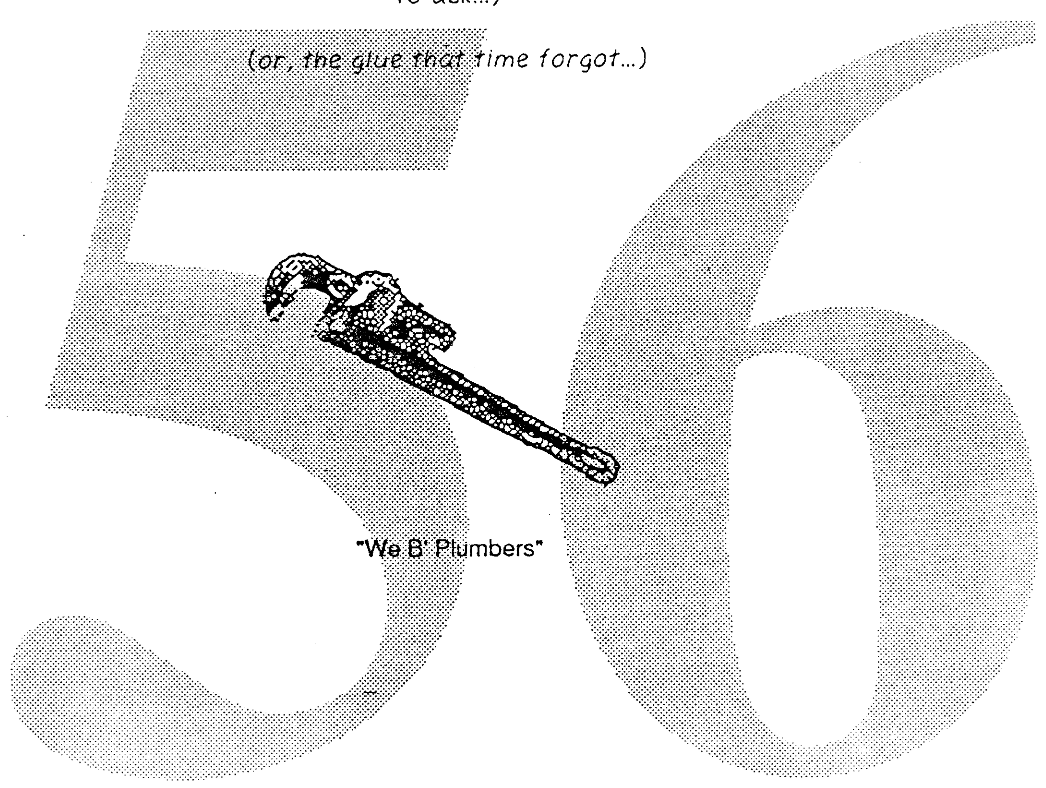
Messages and Pipes

56

The Plumbing

(or, everything you wanted to know about a communications architecture but were afraid to ask...)

(or, the glue that time forgot...)



"We B' Plumbers"

Andy Atkins
(camel trainer)
x49215

56

This section of the Big Pink binder will introduce the roles of messages and pipes in the new communications architecture. The purpose here is to help you *understand* the issues and solutions without getting into the grunge of the implementation. The 411 class descriptions will play that role.

This document assumes that you are already familiar with identities and clients/servers as described in Murf's *Pathfinder* document on naming and locator services and Deb Ortons's *Scream* document on the clients and servers classes.

Introduction

When it came time to integrate networking with the rest of Pink, we had to deal with a number of (possibly contradictory) goals. They were:

- Integrate the network into the system such that networking is as easy to use as interprocess communication (IPC). We wanted to make networking so easy that applications use the network by default.

- Make the network transparent at some level. Communicating with another process, whether it be on this machine or another machine, should require essentially the same code. Ideally, a process should be able to fetch the identity of process it wants to talk to, then use that identity to communicate. Indeed, the application may not even want to know if its collaborators are local or remote.

- Do not force processes that do not use the network to know about the network. Some processes, such as the Layer Manager, are only worried about the local machine name-space and couldn't care less about layers on other machines.

- Do not degrade the performance of local communications. That is to say, the integration of the network with the Pink OS should not adversely affect the performance of local communications as well as the performance of network communications.

Along the way, the Pink Team realized that if we solved the network integration problem, that we could potentially solve the problem for many other folks as well. Both the Finder team and the device-driver folks were grappling with the complexities of naming as well as trying to bridge high-level toolbox access with the low-level mechanics.

So the question was, how can we merge all these worlds and still provide the fastest possible performance? We decided that the place to do this is at the message level. Only upon instantiation of a message does the user need to know who it is they want to talk to. From that point on, the user just plugs and chugs as usual, regardless of the fact that "who" they want to talk to is a process on their local machine or a server on a remote machine.

Architectural Abstract

Given this motivation, the Pink Team decided that Pink really needed a generalized communication architecture. The purpose of this architecture is to provided a consistent yet extensible framework for *all* data communications in the Pink operating system, not just those confined to IPC or those across the network. This goal is achieved by the use of abstract objects that represent the basic building blocks of data communications.

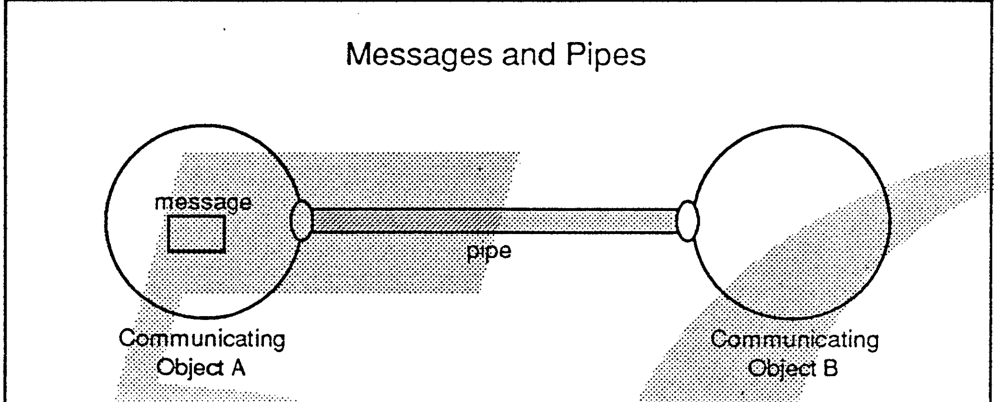


Figure 1. Illustration of Messages and Pipes

These building blocks are:

- *Naming and Locator Services* - A mechanism used to locate a remote entity. See the Big Pink Pathfinder document.

- *Identities* - An identifier handed back from a naming service given a name-space query. Again, see the Pathfinder document.

- *Pipes* - A dynamic representation of an identity. While an identity defines who to connect with and perhaps how to connect, a pipe represents the actual virtual circuit. The pipe manages the communication, including semantics for opening and closing the pipe, keeping track of the pipe's status, and storing ephemeral information specific to the pipe.

- *Messages* - A unit of data that is sent through the pipe. A user will be able to stream information into and out of the message, and call send, reply, and forward methods. A message will actually be an abstract class definition, where its subclasses will implement the actual methods. For instance, an IPC-based message will mix-in the message definition with the TMemory class. On the other hand, a network-based message will mix-in the message definition with the TNetMemory class and provide semantics for creating network data lists.

Architecture

The idea of messages and pipes has its roots in Deb Orton's original architecture for IPC messages. Her IPC wrappers (TKernelMessage) were built on the notion of a transaction, where one end creates and sends a request, the other end receives the request, and then replies, with the requestor finally getting the reply. This paradigm fit in great with what the network needed to do with ATP transactions.

So in defining a generic message interface, we needed to abstract the data storage and data transport. With that, both network and IPC transactions were able to support similar sets of send-receive semantics while hiding the details of data storage and transport.

First step to Nirvana

In order to abstract the data storage, we found it necessary to isolate the messaging semantics in its own class (TMessage). That way, all concrete implementations of a message would need to inherit from both the messaging class and a data storage class (for instance, to send an IPC message, you instantiate a TKernelMessage, which happens to inherit from both TMemory and TMessage).

Isolating the data transport required creating the notion of a pipe. A pipe is the class through which you send a message. That means, for any given pipe, you may have many messages (transactions). In the case of a pipe representing an IPC connection, the pipe may only need to hold the surrogate task for other side. With networking, the pipe may contain address and protocol information. In any case, the transport is completely isolated from the message. TKernelMessage used to have the receiver task intimately bound with the message. With this new architecture, the message knows nothing about the transport, only that it is bound to a particular pipe.

Say a user (such as the client-server classes) wants to make a request of Fred (say). It would first have to instantiate a pipe that knows about Fred (with the help of the Phone Book), and then get an instantiated message from the pipe. Once the user has streamed its request into the message, he can tell the message to send itself. The message in turn submits itself to its bound pipe. Eventually the request goes out and a reply is returned from Fred. When the thread returns control back to the user, the user can then stream out the reply from the message. See, the beauty about all this is that the user doesn't have to change one iota of code, regardless of who or where Fred is. All subclass implementations of a pipe, and all subclass implementations of a message, abide by the base class' abstract method interface.

Beyond transactions

We found that this architecture scales nicely to include communications that are not transaction-based. Say, for instance, a user wants to *stream* to the other side without having to worry about the notion of a transaction. The bytes just "magically" appear at the other end. The messaging semantics already described are transaction-based, so we needed to define another set of classes for a stream-based system.

To be able to easily include packet or datagram communications, communications with local hardware devices, and anything else that requires getting bytes from the toolbox to some lower-level entity, is certainly a big goal of this communications architecture.

The Big Picture

So this brings us to the architecture as it stands today. Figure 2 illustrates the current abstract base class hierarchy.

The TPipe tree defines the abstract base classes for all pipes. The TPipeDataUnit tree defines the set of commands that can be performed on a unit of data bound for the pipe or received from the pipe. Notice that the two hierarchies are rather parallel. This was done on purpose to reflect the fact that a particular unit of data is explicitly bound to the pipe that created it.

Here's a quick run down on each of the classes in the hierarchy:

- *TPipe/TPipeDataUnit* - These are the grandaddy base classes. A TPipe knows how to add and delete its attributes as well as "opening" and "closing" itself. A TPipeDataUnit knows how to alter the attributes of itself and keeps track of the pipe its bound to.
- *TTransactionPipe/TMessage* - These are the base classes for all *transaction-oriented* communications. A user can have one TTransactionPipe instantiated for each server or client, yet have many TMessages. TMessage is the base class from which TKernelMessage and TNetMessage will be based.
- *TStreamPipe/TChannel* - These are the base classes for all *stream-oriented* communications. A user can only have one TStreamPipe per connection and *n* - TChannels, one for each representative band.
- *TDatagramPipe/TPacket* - These are the base classes for all *datagram* (ie.

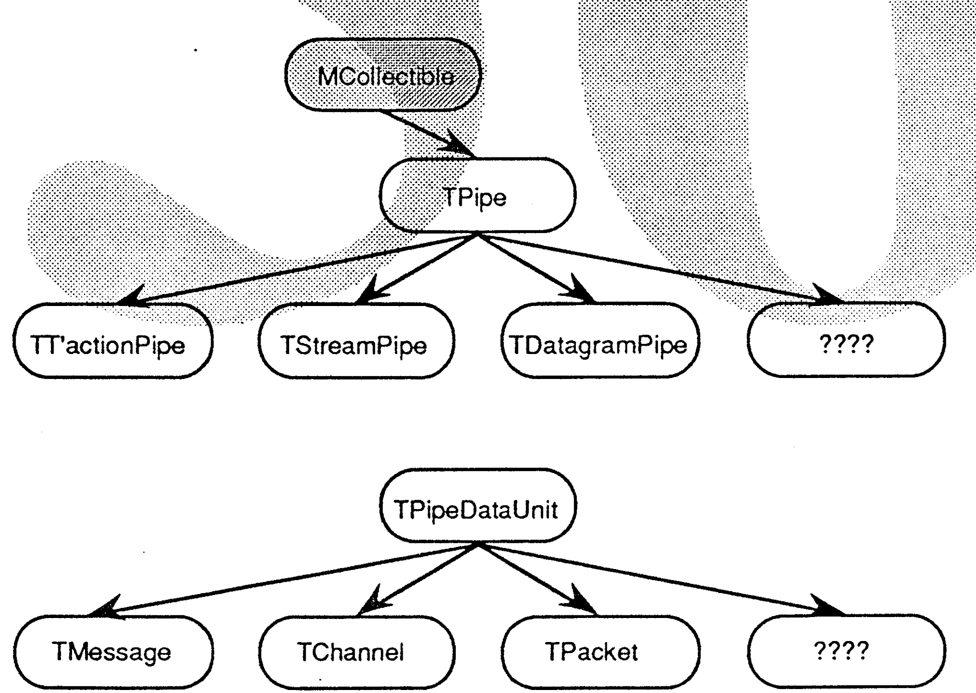


Figure 2. Abstract Base Classes

unreliable!) communications. A user can have one TDatagramPipe per entity and multiple packets.

Pipes

Given the above hierarchy, all pipe implementations share the characteristics defined in the abstract superclass, TPipe. Pipes always represents the communication link. Though a pipe may be implemented as an object containing a surrogate task (in the case of IPC), a network service (in the case of networking), or as a file access mechanism (in the case of the Finder), they all have the following properties:

- *An identity object can return a new pipe.* Any identity, be it a connection-oriented or connectionless identity, will have methods that can create a pipe. In creating the pipe, the identity may be validated (ie. can we really talk to the guy at the other end?) and the pipe itself may go through some sort of initialization sequence. Since either of these actions may fail, the identity cannot guarantee that a pipe can be returned.

- *A pipe will be able to create an identity so you can store it off and use it later.* Since pipes are dynamic in nature (for instance, a user may apply certain attributes to the pipe), the identity representing the pipe may have changed during the course of operation. Because of this, a pipe should be able to construct (and return) the identity that currently represents the pipe. Not only that, a pipe should automagically update the originating phone book entry if the identity was a "specific" identity.

- *A pipe must support a base set of open/close methods.* For many types of pipes, such as a pipe that manages IPCs, opening and closing become NOPs. But in the networking and SCC worlds, opening and closing the pipe become very real concerns.

- *A pipe must support a base set of attribute routines.* These routines may include methods for adding, changing, deleting, and returning the read/write sizes, meta-information (ie. What am I?), and status.

- *A pipe must be able to instantiate a new pipe data unit at the user's command.* Only by doing so can we guarantee an explicit binding between the pipe and the messages it recognizes. See the section below on "Pipe Data Units" for more on this.

Pipe Data Units

Pipe data units, an abstraction of what we've been calling "messages" up to now, allow a user to send and receive data through a given pipe. As mentioned above, these data units are explicitly bound to the pipe it uses. The abstract superclasses (TPipeDataUnit and its immediate subclasses) outline the base set of methods that need to be supported by all concrete implementations (such as TMessage). That way, classes such as MClient and MServer can use any and all messages regardless of its specific implementation.

Data units must support some form of getting and setting attributes on the fly. These attributes in turn temporarily modify the pipe only for the duration of that message. A pipe data unit must also support some form of sending and receiving, but exactly how that is done is up to the abstract subclasses.

Concrete Implementations

From the abstract hierarchy, we were able to design a bunch of concrete classes to solve the communication needs for IPC and the network. Again, the communication architecture is very scalable, so as future communication needs arise, we hope that they fit nicely into this architecture's abstract framework.

Transactions

To date, only concrete implementations exist for transactions using IPC and the network. The concrete hierarchy is illustrated in figure 3.

The abstract interface (that interface which the MClient/MServer classes use) is TTransactionPipe for creating the messages, TMessage for sending, receiving, and modifying attributes, and TMemory for streaming data into and out of the message. TMessage supports synchronous and asynchronous sends, synchronous and asynchronous receives, forwarding, replying, and receives from specific sources.

Example

Here's where we present a conceptual example of what it may look like for an application to get some information from some server. This server may be local or remote; the user doesn't much care.

The first thing the server needs to do is get an identity (however that will be done), and ask the identity to create a pipe. From there, the server must then open the pipe and post a receive.

```
/* PB is a reference to the phone book */
TIdentity serverId = PB.GetId(attribute serverList);
TPipe serverPipe = serverId.GetPipe();
serverPipe.Open();
TMessage& request = serverPipe.GetMessage();
serverPipe.ReceiveRequest(request);
```

The client also needs to create a pipe, where one of the attributes of the pipe is the name and address of the server it wants to talk to. Once the client opens the pipe, it can then stream out the request and block on the reply.

```
TIdentity clientId = PB.GetId(attribute clientList);
TPipe clientPipe = clientId.MakePipe();
clientPipe.Open();
TMessage& request = clientPipe.GetMsg();
TMessage& reply = clientPipe.GetMsg();
data >>= request;
request.Send(reply);
```

The server gets the request, then replies. All the while, the client is blocked until the reply arrives. (There are other ways to do this. The client could have posted an asynchronous request and fetched the reply later, for instance.)

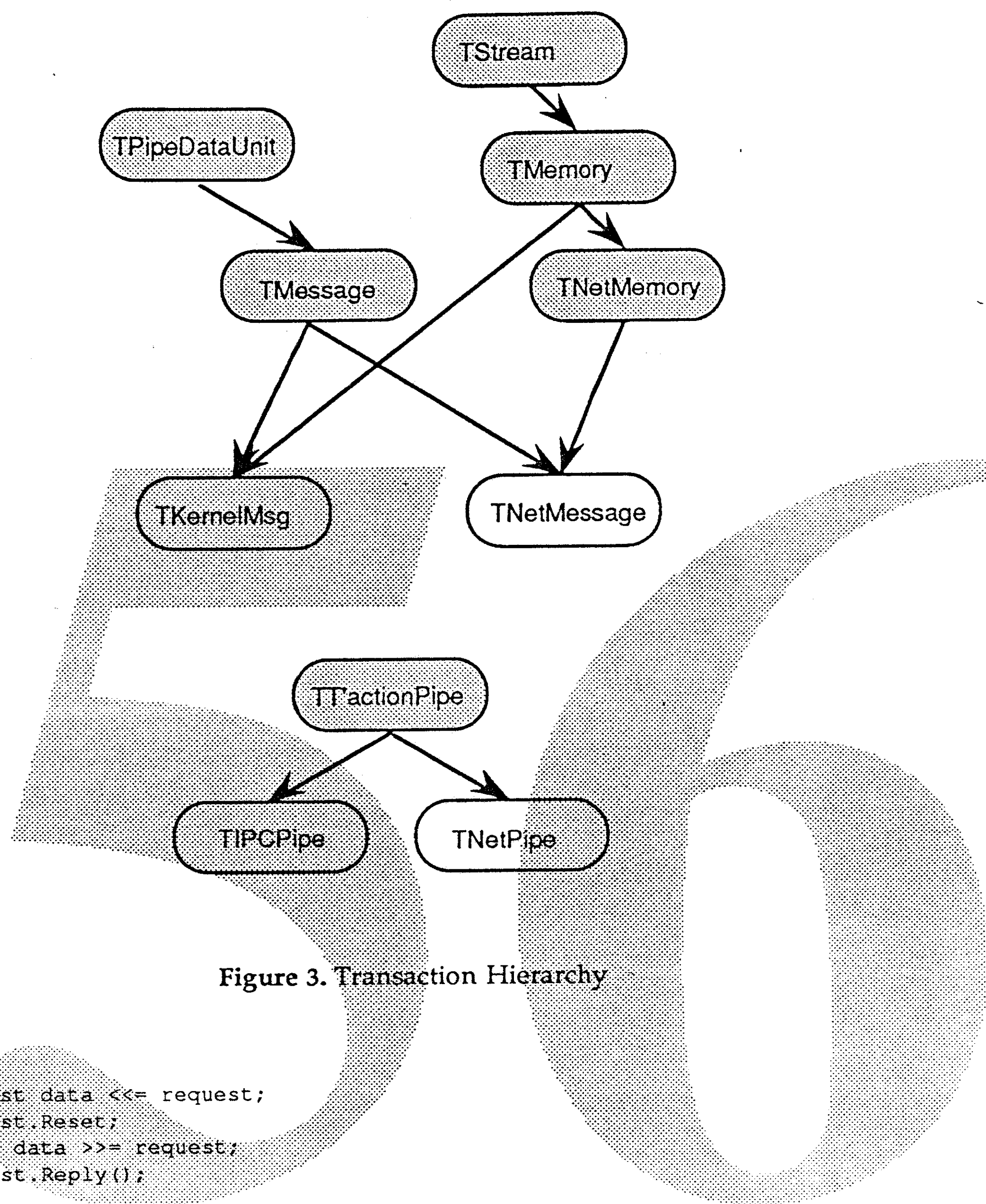


Figure 3. Transaction Hierarchy

```
request data <<= request;
request.Reset;
reply data >>= request;
request.Reply();
```

Finally, the client can stream out the reply.

```
returned data <<= reply;
```

Connection-Oriented Streams

While the stream implementation is still on the drawing board, it is definitely a service needed by the network. Protocols such as ADSP and TCP suggest the use of a stream service interface. Not only that, but I think that we'll discover needs for local stream services, where the exchange of information between processes is *not* transaction-based.

Imagine, if you will, a mechanism by which one process can simply streams into an object, and automatically, the bytes appear in an object in another process' address space (much like pipes work in UNIX). There is no need to call a Send method or a Receive method; the buffers are flushed and filled automatically. (This could be implemented by having the processes stream into a shared memory object that's mapped into both address spaces.)

At any rate, the communication architecture scales to include the concept of a connection-oriented stream. The TChannel and TStreamPipe objects act as front ends to services that are either network-based or local. Subclassed from TPipeDataUnit, TChannel is the object into which you stream, while TStreamPipe, subclassed from TPipe, provides the stream transport.

TChannel, however, is somewhat different from its sister TMessage. A streaming pipe really has no concept of a message or a transaction, so really, there is no way you can have many outstanding messages. Instead, there needs to be a single point of entry through which you can stream. Not only that, but streams often support several "channels" multiplexed across the same connection, where each channel supports a unique kind of data. For instance, a network-based stream may have a data channel, a status channel, and an urgent channel. Thus, with TChannels, the user can at most instantiate as many types of channels as is supported by the protocol.

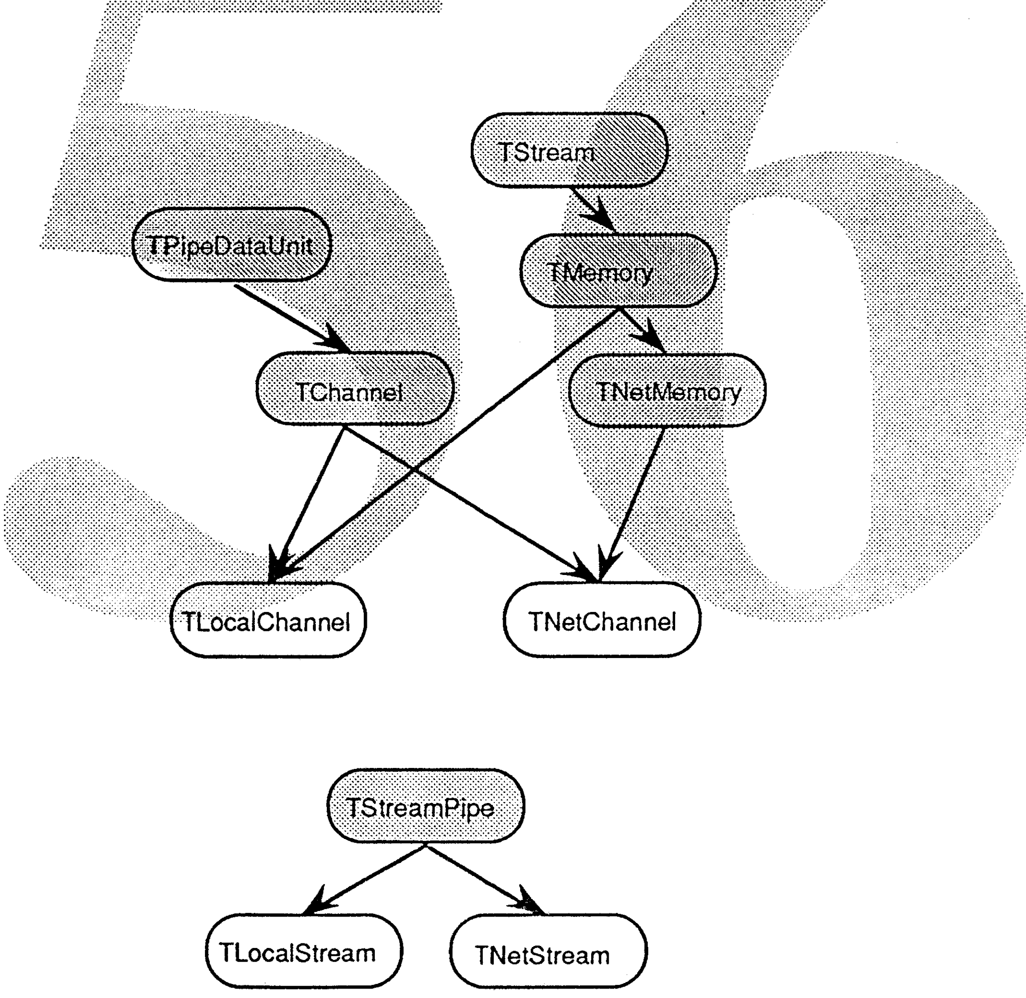


Figure 4: Connection-Oriented Streams Hierarchy

Figure 4 illustrates this class hierarchy.

Datagrams

The construction of an unreliable datagram interface can also benefit from the communications architecture. The network definitely needs a datagram service interface, and (who knows?) there may be other devices or mechanisms that can use this interface too.

Not surprisingly, TDatagramPipe inherits from TPipe to provide the unreliable datagram service. TPacket functions as the object that contains a packet of data. There can be many packets for any given TDatagramPipe. Since there is neither the notion of a connection or a transaction, packets can be sent and received through the pipe in an essential ad-hoc manner.

Outstanding Issues

Since this whole idea of messages and pipes is relatively new, there are still alot of unresolved issues. As people start using messages and pipes (AppleShare, CHER, etc...), some of these issues will come out and be resolved. The issues are listed below, not necessarily in any particular order.

- *Pipe Arrays* - A server (MServer) must be able to listen for requests on more than one pipe (or type of pipe) at a time. One proposal centers around the idea of a pipe array (see our Communications Architecture proposal for all the details), but this remains an unresolved issue.

- *Connection Servers* - Users will need a special type of server, one that does nothing more than accept connection requests, creates a separate connection with the requestor, and then spins off that connection. This function is especially useful for connection-oriented stream in the creation of new connections. This could become a subclass implementation of TStreamPipe.

- *Session Support* - There may be situations where peers will want to communicate using multiple pipes while still maintaining the notion of in-order delivery of requests and replies. AppleShare comes to mind. To make this work, this would require timestamp and session support. Does this belong as yet another type of pipe, or as some entity that uses pipes and client/servers?

- *Pipe Creation* - With all the discussion going on with "desktop objects" and "phone books", it is still unclear how these pipes will get created, and who's responsible for creating the identities from which the pipes are derived.

- *Death Notification*- It is still unclear how we are going to add death notification of the pipe architecture.

56

SCREAM

Server/Client

56

Server/Client Services

Deb Orton and Irazu de Colorado

56

Server/Client Model

The server client classes are intended for use in cases where two separate entities need to communicate (and possibly exchange data or objects) using request-based transactions. These entities may or may not be on the same team, may or may not use IPC (Inter Process Communication) and may even be on separate CPU's connected through some network. There will usually be only one instance of any server and multiple instances of the associated client. Client and server objects may be intermixed with other objects and even with different subclasses of themselves (i.e. a single object may be a client of more than one server, and a server may be a client of some other server). The exception to this is that any given object may only inherit or contain ONE server. No mechanism is currently provided for handling multiple servers in a single object.

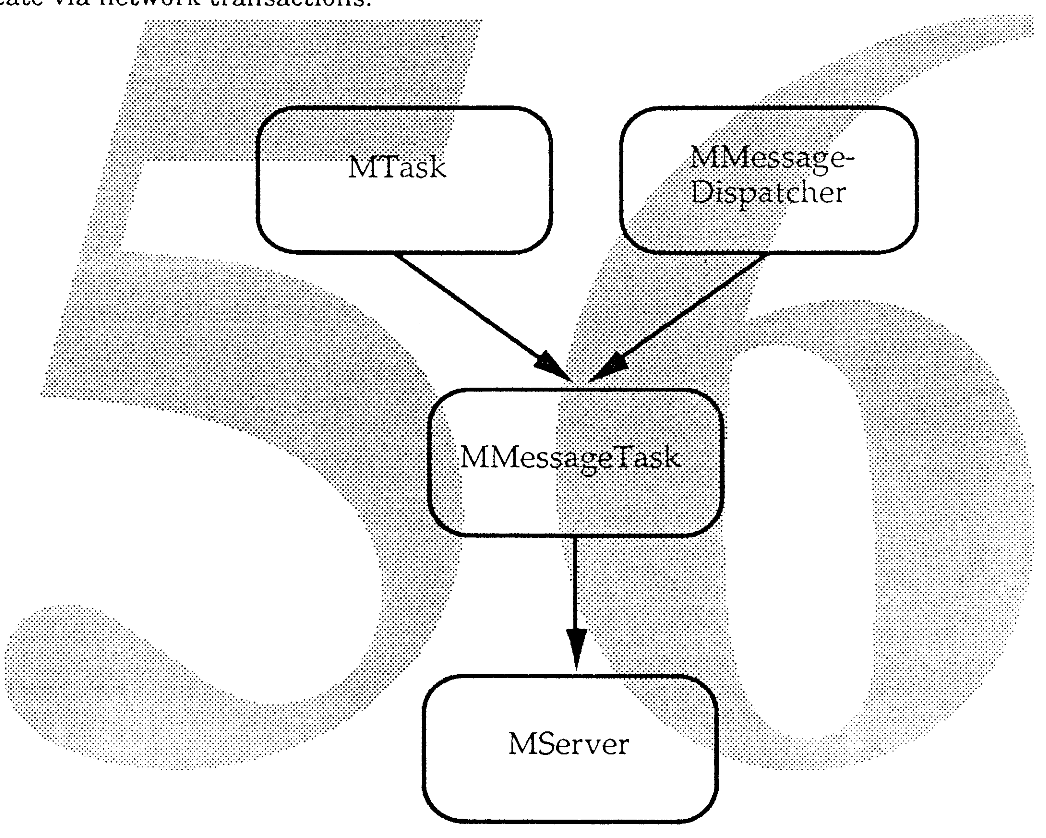
Clients are not multi-threaded and by default are not multi-thread safe. Servers are also not multi-threaded and process requests one by one. Asynchronous handling of requests may be accomplished by spinning up a light-weight task to process a given request in parallel.

Servers and clients communicate via transactions. In the case of both objects on the same machine, IPC is used. For objects on different CPU's, network transactions are used. Developers may choose between same-machine server/clients and network-aware server/clients. The overhead for transactions is slightly higher for network-aware servers/clients although the programmatic interfaces are the same whether going over the network or locally.

Servers

The server object is a light-weight task that blocks waiting for a request to arrive, processes the request, and then goes back to waiting for another request. Requests may be handled asynchronously by spawning a task to handle the request. Mechanisms are provided for handling exceptional circumstances (client died, out of a resource etc.). Very basic servers consist of a method for each request the server may handle and registration of those methods in the constructor. Complex servers may also do special exception handling, spawn light-weight tasks to do asynchronous processing and even communicate with another server.

Servers on the same team as their client(s) communicate via IPC (TServerMessage), but do not copy section information via kernel calls, rather reading and writing is done by directly touching the appropriate addresses. Servers on separate teams from their clients communicate via IPC, sections and possibly segments (TServerSegment). Servers and clients on separate machines communicate via network transactions.



MServer

Methods for General Consumption:

The **Constructor** creates a server object. This can be done either as a task or as a team. (See the Wrapper's documentation for more information on tasks and teams.)

RegisterRequest must be called from witin a sub-classes' constructor (or some initializer). It registers the method name as a possible request for that server. Two flavors are provided. One registers the request to be handled synchronously (by the server itself); the other registers the request to be handled by a server helper task.

GetDataStream returns a pointer to a data stream to be read or written. The data stream is NOT part of the message text and may generate a section to be sent with the message. This

stream may or may not be copied by setting the "buffered" flag. If buffering is FALSE, then every read and write will generate a kernel call to preform the operation. If "buffered" is TRUE, then the stream is copied into the current address space and reads and writes do not generate kernel calls. This stream grows automatically, as needed.
GetMessageStream returns a pointer to a message stream which may be read or written. The message stream is part of the message text and is COPIED with the message when sent or returned. This stream is fixed length (approximately 120 bytes).
Stream Operators write and read the task object piece of the server.
HaltServer tells the server to exit. Call this method when you want to shut down the server.

Methods for Getting Information:
GetCurrentRendezvousID returns the rendezvous identifier associated with a message and its sender/forwarder. RendezvousID's are unique numbers system-wide.
GetCurrentClient returns a surrogate task object representing the current client task.

Methods for Extending Behavior:
HandleDefaultRequest is called when a client makes a request with no specific request string. Override this method if your server only does one thing.
HandleUnknownRequest called if an unknown request comes in.
HandleReply is called when an in-coming message is a reply instead of a send. By default the message is thrown away.
HandleSenderDied is called when the server attempts to reply to a client that is no longer valid. Override this method to do something special. By default the response is thrown away and the server continues waiting for requests.
HandleException called if an exception occurrs.
ReleasePendingRequest allows the implementer to control whether the server handles the request or passes it on to a server helper task.
CreateServerMessage instantiates the appropriate message object for communication between the server and the client. Override it if you wish to change the server message object. Create your message on the heap (using new) and return a pointer to it. The server handles deallocation.

Methods Used for Implementation:
HandleServerMessage implements the servers main loop. It sits waiting for a request to come in, processes the request and then goes back to sleep waiting for another request.
SignalNeedBiggerSection what to do if the section is too small.
SetMessage & GetMessage are used internally to set/get the message.
Initialize is used internally to initialize the server.

MServerHelperTask

Methods for General Consumption:
The **Constructor** creates a server helper task object. This task object sits in a loop waiting for reqeusts to be forwarded by the server. By default, there is only one task per request type and subsequest reqeusts are queued for the helper task.
GetDataStream returns a pointer to a data stream to be read or written. The data stream is NOT part of the message text and generates a section to be sent with the message. This stream may or may not be copied by setting the "buffered" flag. If buffering is FALSE then every read and write will generate a kernel call to preform the operation. If "buffered" is TRUE, then the stream is copied into the current address space and reads and writes do not generate kernel calls.
GetMessageStream returns a pointer to a message stream which may be read or written. The

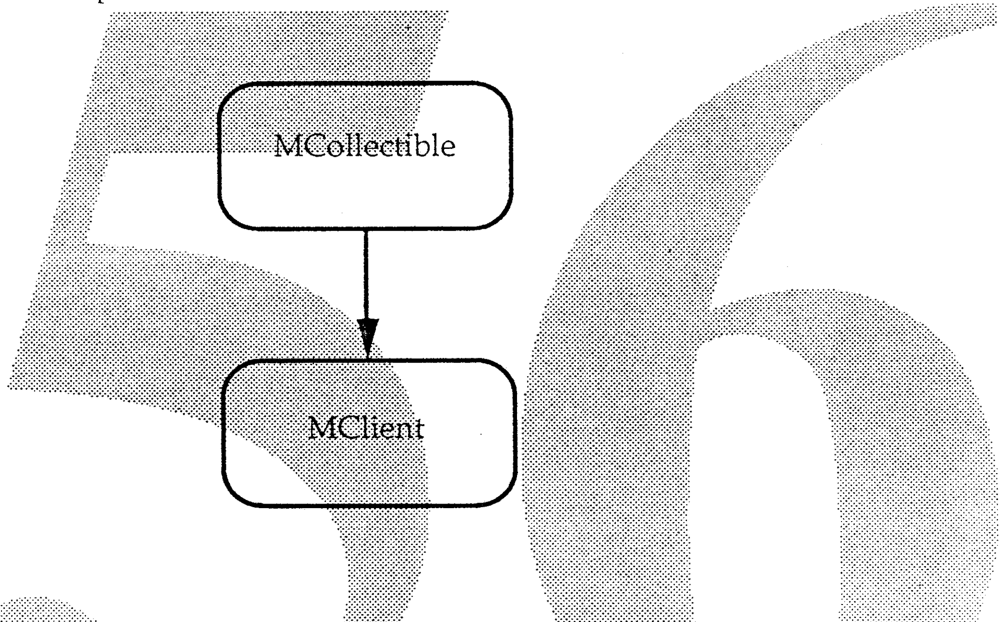
message stream is part of the message text and is COPIED with the message when sent or returned.
HandleServerRequest should be overridden to implement the handling of the request.

Methods Used for Implementation:
HandleServerMessage implements the helpers main loop. It sits waiting for a request to come in, processes the request and then goes back to sleep waiting for another request.

Clients

A client object may be mixed-in, be a private base class or simply a member of an object. Simple clients need only specify the request, package any data, and send it to the server. Complex clients may communicate with more than one server, generate asynchronous requests, or communicate through shared memory (TClientSegment). Each task should have its own client for any needed server, or the applications programmer will need to provide concurrency control within the client object. Mechanisms are provided for handling exceptional situations (the server died, out of needed resource, etc.). By default, clients start their respective servers in a separate address space if they are not currently running when the client sends its first request.

Clients communicating to servers in the same address space provide some optimization for information transfer. Clients on the same machine, but in different address spaces communicate via IPC and sections (TSections) -- and possible shared memory (TServerSegment and TClientSegment). Clients on separate machines from their servers communicate via network transactions.



MClient

Methods for General Consumption:

Constructor creates the client object.
StreamOperators write the client information to and from a stream.
SendRequest fires off a request to the server and waits for a reply. A data stream and/or a message stream may be sent.
AsyncSendRequest fires off a request to the server and then continues. The reponse must be handled at a later time by calling WaitForResponse or CheckForResponse.
OneWaySendRequest fires off a request to the server and contines. If a section needs to be sent with the request then OneWaySendRequest will return a new segment for the programmer to use.
GetDataStream returns a pointer to a data stream to be read or written. The data stream is NOT part of the message text and may generate a section to be sent with the message. The data stream is grown automatically, as needed.
GetMessageStream returns a pointer to a message stream which may be read or written. The message stream is part of the message text and is COPIED with the message when sent or returned. The message stream is fixed length (approximately 120 bytes.)
CheckForResponse checks if there is any response fromt the server waiting to be processed.
WaitForResponse blocks until a response from the server arrives.
ShutDownServer sends a request to the server to shut itself down.

Methods for Getting Information:
GetServer returns a surrogate task representing the server object.
GetServerName returns the filename of the server.
IsServerKnown returns whether or not the server task is known.
GetCurrentRendezvousID returns the current rendezvous ID.

Methods for Extending Behavior:
HandleServerDied what to do if the server dies.
FindTheServer how to locate the server (by default it uses the nameserver). If the server is not found, then this guy calls StartTheServer.
StartTheServer how to start the server. OVERRIDE this guy if you want to start the server as a task (not as a team). You will need to create your server object and then start him up.
SetRequestPriority sets the priority for a request.
Set/GetDeleteServerOnFree tells to client if the server object (may be a surrogate) needs to be deleted when the client is deleted.
SetServer is used to set the server task explicitly.
SetServerName sets the server file name.
CreateServerMessage instantiates the appropriate message object for communication between the server and the client. Override it if you wish to use your own server message object. Create your message on the heap (using new) and return a pointer to it.
SetMessage & GetMessage are used to set/get the message.
SectionUsed returns whether or not a section was used to send the request.

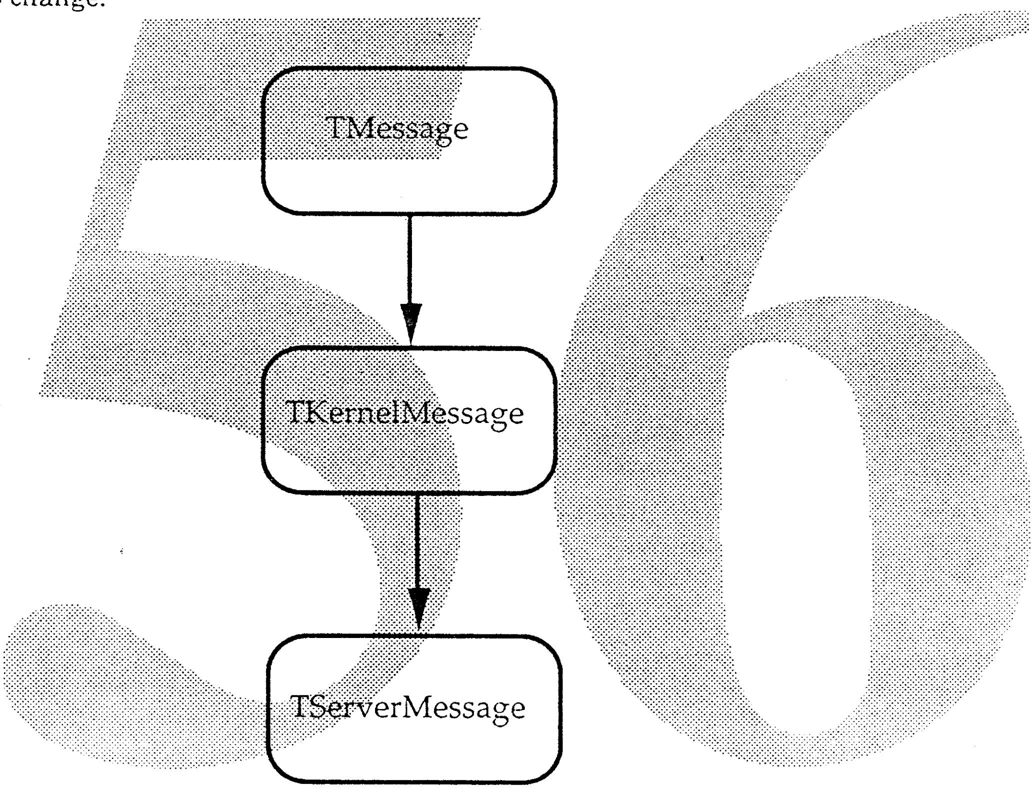
Methods Used for Implementation:
Initialize initialies the client object. This guy checks to see if the server task is known. If not, he calls FindTheServer.
HandleNeedBiggerSection what to do if the section is too small.
HandleDataPackedInHeader what to do if the data stream was packed in the message.

ServerMessages

The server message object (TServerMessage) provides communication and transfer of data between the Server and the Client. It provides a protocol for transactions between servers and clients. Specific implementations of this class may allow communication across networks, via IPC or possibly through shared memory.

The server message inherits from the basic message object (TMessage) and adds some server/client specific protocol. Server and client objects are unaware of the transport mechanism used to handle their requests and responses and operate only on the protocol defined by the server message object. Any special "magic" to move transactions across networks, etc. is implemented in a subclass adhering to the server message protocol.

This class is still in the design and prototype stage to support transparent networking and is subject to change.



TServerMessage

Methods for General Consumption:

The **Constructor** creates a message object for the server and client to use.
The **Stream Operators** write the message information to and from a stream.
GetDataStream returns the section associated with the message.
GetMessageStream returns the available portion of the message text (the piece not reserved for server/client communication information).
GetSurrogateDataStream returns the surrogate section associated with the stream.
Reply handles copying the data stream into the message (if possible) and writing any unbuffered surrogate sections back to the real section.
Reset resets the message and all associated streams.

PackData is available to clients and servers to facilitate the packing of a request and its data.

Methods Used for Implementation:

UseDefaultRequest looks at a flag in the message to see if a request is being passed.
SetDefaultRequest sets at a flag in the message to signify that no request is being passed.
GetUserStart returns the starting address of the message available to the user.
GetUserLength returns the length of the message available to the user.
GetMessageLength Returns the number of bytes available in the message text.
GetRequestLength returns the length in bytes of the request id.
PackRequest and **UnpackRequest** are used internally to pack and unpack the request id.
SignalNeedBiggerSection generate an exception stating that the section is too small for the response.
HandleNeedBiggerSection handle the exception that the section is too small for the response.
HandleDataInMessage what to do if the data stream has been packed in the message.
SmartOneWaySend creates a new data stream if the old one is used for the one way send.
DataInMessage looks at a flag in the message to see if data was packed (by the client or server) in the message. The programmer can explicitly pack data in the message by using GetMessageStream(). If the user did not pack information in the message stream and wants to pass data in a section that will fit in the message text, then it is copied in and this flag is set.
SetDataInMessage set the flag to signal that the data stream has been packaged in the message.
NeedSection checks a flag in the message that signifies that the section currently available is too small. Unpack in the size needed.
SetNeedSection Set a flag in the message that section currently available is too small. Pack in the size needed.
ClearFlags reset all the internal flags (NeedBiggerSection etc.)

Example

Suppose I wish to build a random number server. The purpose of this random number server is to generate a pseudo random number for a client and also to allow a client to set the seed used for the number generation. This will be accomplished by building the following objects:

1. TRandom

   This is an abstract superclass that defines the protocol for the random number generator. Both the server and client objects inherit from it and adhere to the interfaces defined here.

2. TRandomServer

   Only one instance of this guy is created (during startup of the system or some such). He registers himself with the global name server and then goes to sleep waiting for a request. There are three methods to handle the 3 possible requests: **SetSeed**, **GetRandomNumber**, and **SetSeedAndGetRandomNumber**. When a request arrives, the appropriate method is called and may or may not send some response data back to the client. In the case of **SetSeed**, no response data is sent, for **GetRandomNumber**, the new random number is sent back to the requester. The third requests uses the seed that was sent, generates a random number and returns it to the client.

   An example of processing a request asynchronously is given (using a "helper task"), although the extra task is not necessary in this example. (Just tryin' to be helpful!)

3. TRandomClient

   Multiple instances of this guy will most likely exist. Any object that wishes to get a random number will simply mixin this object. The client object knows how to talk to the server, and what requests the server understands. The client packages the request and any associated data and sends it off to the server. When the server replies, the client unpacks the response and continues.

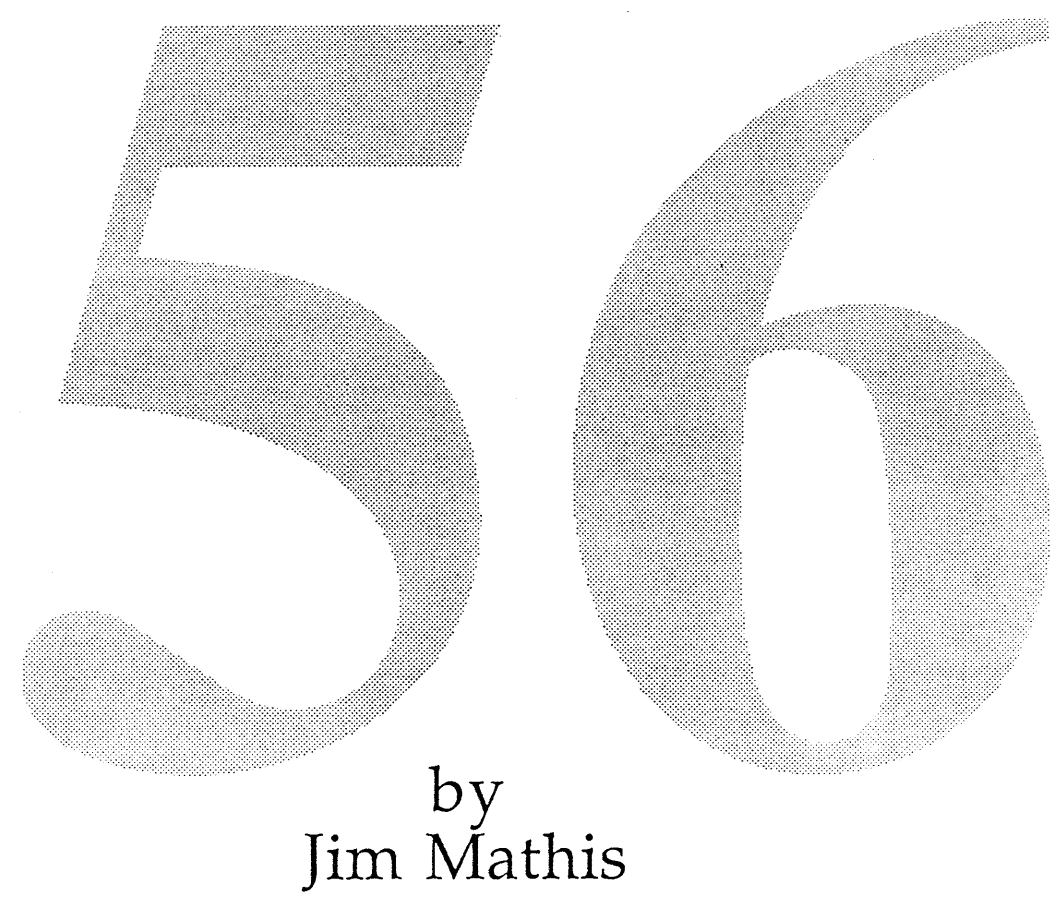
N.B. This example is taken from code that I use to do testing and therefore attempts to run through multiple paths of the code. The typical developer would not usually use such a variety of paths (AsyncSends, message AND data streams, helper tasks etc.) for no good reason.

56

RedEye

Pink AppleMail™

56



by
Jim Mathis

56

Overview

RedEye provides electronic messaging interconnection with the AppleMail system being developed for System 7. Since AppleMail for Blue is currently a few weeks away from Alpha, it is difficult to exactly define what it will look like on Pink. However, we expect the following to be true:

- The existing Blue AppleMail software will probably not be able to run inside the Blue Adapter, requiring a rewrite for Pink.[1]
- Pink will initially support only the message client software and depend on the Blue implementation of the message server (running in a separate machine) for store-and-forward routing.
- RedEye will be integrated into the Pink Finder framework in much the same manner that 20/20 is integrated with NuFinder. The actual user experience may be different on Pink owning to the difference between the Pink and Blue Finder/Desktop.
- The same mail protocols developed for Blue will be used on Pink to ensure interoperability.
- Program-to-program message communications will be supported in addition to the user-level EMail functions.
- The Personal Gateways built for Blue will probably not work with RedEye. Issues surrounding the set of Personal Gateways that must be built and how they are implemented is as yet undefined.

The design and implementation details presented are subject to change as both the Pink world and AppleMail evolve.

Architecture

In Pink, what is often though of as a single function--Electronic Mail--is actually decomposed into two separate but related services: non-immediate, store-and-forward delivery of data, called messaging; and an interface that let users manipulate messages intended for human consumption, called mail. On Blue, 20/20 take a first step at creating this separation; in Pink, we are seriously looking at what network services mean when the data is not delivered immediately upon transmission.

Store-and-Forward Networking

The foundation of electronic mail is a reliable delivery service that typically has characteristics different from other types of reliable transport protocols. The most important distinction is the time-frame over which the data is delivered. To allow for the interconnection of different mail systems, an intermediate store-and-forward step is often required; the time it take to complete delivery to data to the destination can range from seconds (for machines sharing a direct network connection) to perhaps tens of days. During this time, the sender machine may crash or be powered-off or otherwise lose memory state.

---

[1] With the Blue Adapter effort restaffed, we will explore the possibility of running NuFinder extensions within the Blue Adapter as an avenue for quick delivery of AppleMail capability for Pink.

The non-immediate delivery of data has a variety of benefits that can outweigh the inconvenience of the time delay involved. When time is not important, data can often be transmitted at off-peak hours for lower communication charges. Simple economics may prohibit full-time network connection between users, particularly between users geographically distant. Interconnection with other messaging systems often go through gateways that add delay. For personal computers users, particularly users of portable machines, there may never be a time when both the sender and recipient machines are powered-on and connected to the network; the store-and-forward of data by a third-party agent would thus be required.

Messaging in Pink

The ability for a program to create, send, and receive store-and-forward messages will be provided by extensions (subclasses) to the client/server framework described earlier. Additional methods will be provided for the manipulation of the waiting message queue that can be called by client/server applications that are aware of their use of AppleMail messaging. Program to program communications such as that provided by the System 7.0 PPC toolbox will be handled using these extensions to the basic client/server classes.

Pink will also provide an authentication toolbox that allows programs to build authenticated streams over ADSP as well as transactions over ASP/ATP; the network wrappers will handle invocation of the authentication toolbox automatically. RedEye will use these services to provided secure pickup and delivery of AppleMail in a manner equivalent to that used in Blue.

Interface to Messages for Human Consumption

Since much of the user value in a mail system is its direct communication to human users, we consider this a special case of the more general uses of messaging. The Pink electronic message service will be provided through a set of Pink Finder desktop objects built to reference various lower-level toolbox objects. This architecture provides for both user-level (i.e. from the Finder) electronic messaging, as well as support for programs that which to have embedded messaging capability.

The directory access functions provided by the ADAS client for AppleMail in System 7 will be completely integrated into the PhoneBook by which users access all network services. The details of the PhoneBook and accessing of services are described in the PathFinder and Pink Finder documents.

RedEye will provide for the development of Personal Gateways to handle access to foreign mail systems in a manner similar to that used in 20/20. The development of a Personal Gateway for RedEye will also entail the construction of directory naming class objects that will be called by PathFinder and the Phonebook. The PhoneBook always acts as a central repository of naming and addressing information in Pink independent of the lower-layer networking or communication protocols involved.

Open Issues

Since RedEye is in such an early specification state, there are many open issues to be resolved. This section attempts to list the non-obvious issues that need to be considered:

1) Overall, how can mail and messaging be better integrated into the Pink to provide user functions not possible under Blue?

2) What is the interaction between RedEye and the Pink Scripting system? How much scripted manipulation of the user-level mail messages should be provided?

3) What extensions might be required for mailing large video images when running on a Jaguar?

4) Can Cher use messaging to maintain shared documents between two or more users that do not have a direct network connection?

56

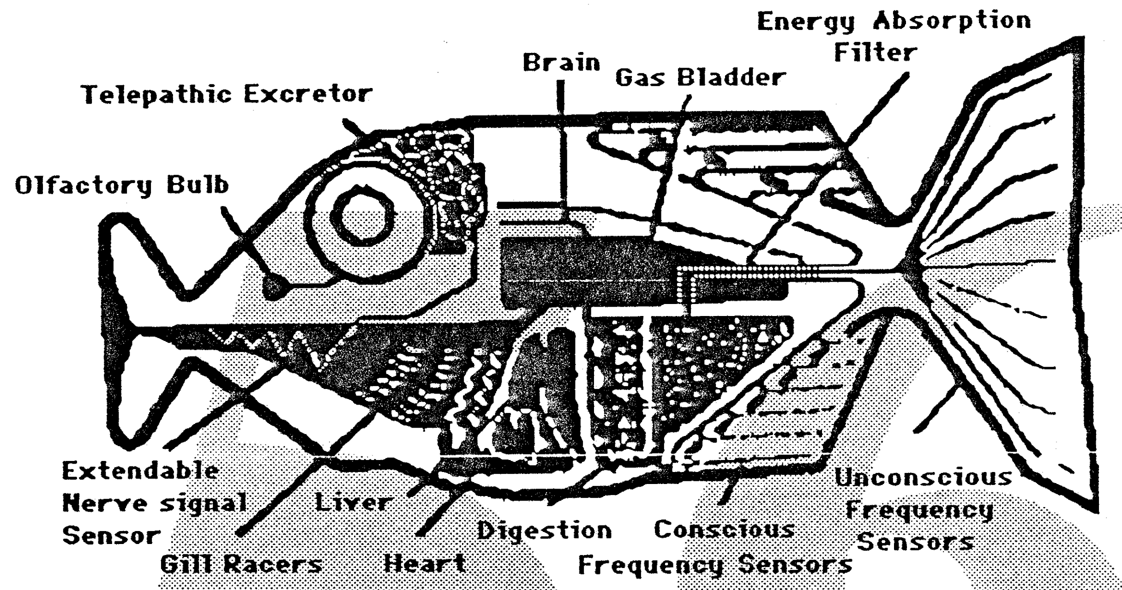
56

BabelFish

The Network

56

BabelFish



"The BabelFish is small, yellow, leechlike, and probably the oddest thing in the universe. It feeds on brain wave energy, absorbing all unconscious frequencies, and then excreting a matrix formed from conscious frequencies and nerve signals picked up from the speech centers of the brain, the practical upshot of which is that if you stick one in your ear, you can instantly understand anything said to you in any form of language."

- excerpt from 'The Hitchhiker's Guide to the Galaxy'

compiled by
Andy Atkins

Jim Mathis - Program Lead
Andy Atkins
Anne Muller
Jim Murphy
Mike Quinn

56

The BabelFish project is responsible for providing easy-to-use, flexible, and extensible network objects on Pink. BabelFish's higher layers provide a collection of network wrappers where applications and other services can easily and uniformly plug into the network. The lower layers implement multiple protocols and links within Pink's object-oriented framework while providing the hooks and abstract classes for future protocol/link implementations.

This document will provide an overview of the BabelFish architecture. For more detailed descriptions, please review the *BabelFish ERS* (coming soon to a store near you) and the 411 documentation. This document assumes that you are already familiar with messages and pipes, as described in the Big Pink *The Plumbing* document. It also assumes that you are familiar with identities, as described in the *Valhalla* and *Pathfinder* documents.

Introduction

The BabelFish project is providing the basic networking support for the Pink operating system. We are doing this by building a networking object hierarchy that is flexible enough to support multiple protocols, network links, and accessing methodologies. This hierarchy is not tied to any one protocol like AppleTalk, or to any one view of networking, such as client-server model. Rather, it is a set of tools that will allow developers to choose the level of abstraction and implementation that bests suits their needs.

Main Goals

The BabelFish project intends to provide a first class object-oriented application programming interface (API) for networking for Pink. Implementation of this goal is centered on the following concepts:

- Simplicity: Applications that only require simple networking services, will only require simple programming by developers using our API.
- Flexibility: The Pink API will provide the ability for network developers to program by extension where the standard functionality of the API is unsuitable.
- Extensibility: The API will be built upon an abstract framework which will allow developers to include new types of networking services that were not originally anticipated.

All of the goals mentioned above can be achieved by using the techniques of object-oriented programming and by providing an abstract calling interface for developers of applications that will be using networking services.

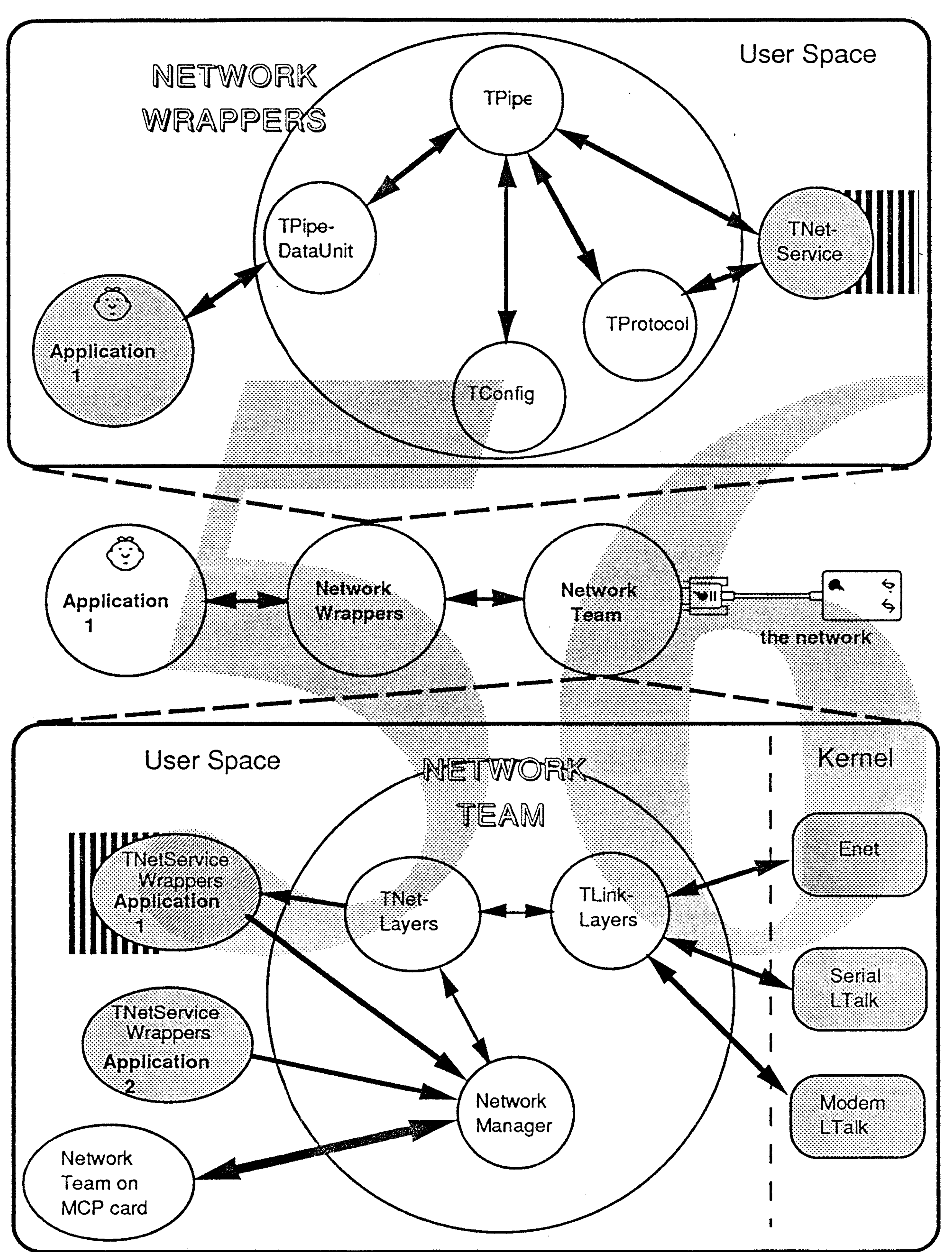


Figure 1: Network Object Organization

Architectural Overview

The network architecture is functionally split into two separate pieces. As illustrated in Figure 1, most applications communicate with the network wrappers. Existing in the application's address space, the wrappers provide a single consistent interface to the network while hiding the peculiarities of the protocol machinery. In using pipes, identities, and messages, the application may not even know that it is using the network.

The protocol and link layer machinery, however, exists within the network team where they can be shared across all applications. The wrappers communicate with the network team using network requests constructed in a shared memory address space. Applications using the network are protected from each other since the network manager allocates a separate distinct shared memory segment for each user of the network. This manager also has the responsibility for multiplexing outbound messages to the appropriate protocol module and inbound messages to the correct shared memory address space.

Mix And Match Protocols

In designing the low level classes, we decided that the ability to mix and match different layers in a protocol stack was very important. With this criterion, we no longer implement protocols as monolithic 'stacks'. Instead, we implement each protocol individually and then dynamically bind them to the network layer with which they will ultimately communicate. With this mechanism, we are able to instantiate a path that allows, for instance, ADSP to sit on top of IP.

These protocols operate in one central team per CPU. This allows all network data to be addressable by all of the network layers without resorting to IPC (except for communications to and from the application address space) or to network teams running on other CPUs.

Service and Protocol Autonomy

In contrast, the network wrappers don't so much implement protocols, but instead hide the protocol grunginess from the application. The first way it does this is by providing applications a common service interface to the network. For instance, if the application required a stream-based interface to the network, regardless of the protocol used to carry out the service, it would ask the wrappers to provide an instantiated channel to which to stream. By doing so, the application does not lock itself into a particular protocol. In fact, the application may later on specify a different protocol while still interfacing with the same set of pipe and message methods. In practice, however, the protocol used in accessing a remote service will generally be stored in the identity object.

The second service provided by the network wrappers is to prepare all outgoing packets for the network team. With its knowledge of protocols and the configuration information provided by an identity, the network wrappers can automatically create network requests and add in the appropriate attributes.

The application does not instantiate the network wrappers directly, but instead uses the methods provided by a phone book or desktop object to create identities. In turn, these identities dictate the behavior of the network-based messages and pipes. A network pipe constructor, for instance, creates the appropriate subclass of the protocol wrapper object and sets up the protocol path in the network team, all based on attributes in the identity.

One of the driving forces in our design was to have the low level classes stand as an autonomous implementation of networking under Pink. Meaning, the network team serves the network wrappers but is not dependent on the wrappers. That way, the Blue network adapter or "critical" network applications have the ability to access the network via a special object (called the network team service interface) in the low level classes as just another client. Though this option may deliver faster throughput, the user of this object must do a lot of grungy preparation (such as instantiating the appropriate class of protocols and links) and fill in all the protocol-specific attributes of a network request. Because of these drawbacks, we believe that most applications will access the network via the network wrappers.

Network Requests and DataLists

The BabelFish architecture is centered around the network request object (TNetworkRequest), which acts much like a suitcase for requests that travel the network. Just like a suitcase, it sports compartments for different types of information (called attributes). Attributes include addresses, channels, protocol types, interval, retry, and so on.

Both pipes and identities use attributes. The network implementation of pipes, for instance, builds network requests for submission to the network team. It does this by creating a network request from the user data, then adding in the attributes defined in the message and identity. It also has the ability to figure out which attributes are appropriate for a given transaction so that the network team is not overwhelmed by a huge and possibly irrelevant attribute list.

Network objects within the network team, however, pass dataLists (TDataList) instead of network requests. Datalists are essentially network requests with fields for supporting headers and trailers as well as additional completion information[1]. A particular network layer processes a dataList by adding or replacing the appropriate attributes, headers, and trailers within the dataList. A crucial property of the dataList is that it allows for the mixing and matching of different layers without locking out future networking protocols and services. Unlike standard parameter blocks where the submitter and receiver must agree on both the structure and content of the block, our network layers need only agree on the structure of a dataList. If there exists an attribute in the dataList that the layer doesn't understand, it simply ignores it and passes it through to the next layer in the chain. Once the dataList reaches the end of the chain (usually in a link layer object), the object converts the dataList into a packet and sends it over the link.

Abstract Hierarchy

In the sections that follow, we will briefly outline the abstract hierarchy that makes up the network wrappers and the network team. We will also summarize the data flow that takes place in both the wrappers and the team.

Interface Classes

The interface classes encompass those classes that are accessible to objects outside the network team. These classes are:

---

[1] In fact, the TDataList class descends from TNetworkRequest.

- TPipe: The TPipe class defines the object through which an application can submit data to the wrappers and receive data from the wrappers.

- TPipeDataUnit: The TPipeDataUnit class encapsulates the user data for the wrappers.

- TNetworkRequest: The TNetworkRequest class implements the semantically-tagged parameter blocks used to process all network requests. It is the object with which the wrappers build data to send to the network and receive data from the network.

- TAttribute: The TAttribute class implements an 'attribute'. It contains a tag indicating what the attribute is for, another tag indicating the owner of the attribute, and optionally some data indicating the value for the attribute. A TAttribute object is never created directly. Instead, users instantiate one of the type-safe subclasses (i.e. TNumericAttribute, TMillisecondsAttribute, TTextAttribute, TTextArrayAttribute, etc.).

- TAttributeList: The TAttributeList class implements a list of TAttributes.

- TNetService: This class implements the network team service interface for the network. It corresponds to a 'route' through which a network request must travel. One example of a route might be a standard PAP workstation implementation: PAPWorkstation - ATP - DDP - Ethernet. TNetService's sole interface is the *ProcessRequest* method for processing a TNetworkRequest object.

Network Wrapper Classes

The goal of the high level network wrappers is to provide a standard common interface to all networking services. This goal is accomplished by separating the interface for acquiring networking services from the implementation of that service. By separating the service interface from a specific protocol implementation, a client application will not have to special case each different type of protocol the application may be using or limit that application to be able to use only one protocol.

The wrappers accomplish this by implementing a service interface based on the pipe and message architecture. Though its service interface may be pipes and messages, underneath, it requires several classes for configuration, inbound message caching, and protocol management. In fact, the configuration classes help separate the pipe service from the protocol it uses by dynamically binding the protocol object with the pipe at connection-creation time.

The following is a brief description of each of the classes that are a part of the high level wrappers:

- TProtocol: The TProtocol class is where pipes and messages interface with the protocol machinery (affectionately called "where the rubber meets the road"). It is the responsibility of the TProtocol class to build into the network request the appropriate attributes and commands given the protocol involved.

- TWrapperTask The TWrapperTask class helps the pipe manage inbound message from the network that have not yet been requested by the pipe's user. We need to internally queue inbound packets so that we don't tie up the network team's precious resources.

- TConfig: Network-based pipes use the TConfig class to instantiate the appropriate subclass of TProtocol and create the correct route request for the network team service interface.

Network Team Classes

The low-level classes provide the nuts and bolts interface of the network system. There are two categories of low-level classes: 1) the network team classes which provide routing and interface services, and 2) the protocol/link classes which implement the functionality of the network (i.e. protocols and links). While it is possible for applications to use these low-level classes directly, the use of the network wrappers makes it much easier to use the network. Using the low-level interfaces directly should only be done where speed is of the essence and developers know exactly what they're is doing (fondly called "driving without seatbelts").

The routing and interface classes used within the network team are below:

- MNetSharedMemory This class implements the interface to the shared memory used by both the application client and the network team. It is basically a static object which manages the memory pools.

- TNetMgrClient This class implements the interface to the Network Manager. It contains methods to control the binding with each client application, as well as maintains a small client data base so that the network team can clean up if a client dies.

- TDataList This class implements the object which is passed from layer to layer to propagate requests.

The protocol/link layer classes contain the code that implements the protocols and link connections. Listed below are brief descriptions of those classes specific to the protocol/link layer classes:

- TNetLayer This is the parent class of all network layer implementations.

- TLinkLayer — This is the parent class for link layer implementations. It is a subclass of TNetLayer.
- TDatagramLayer — This class defines the generic interface to a packet-oriented datagram layer.
- TTransactionLayer — This class defines the generic interface to a transaction-oriented layer.
- TStreamLayer — This class defines the generic interface to a stream-oriented layer. It has semantics for both transaction and stream data transfers, as well as connection establishment.
- TNamerLayer — This class define the generic interface to a naming protocol layer such as NBP/ZIP.

Data Flow

In this section, we will outline the flow of data in both the wrappers and network team. This will include object instantiation, opening and closing of connections, and sending and receiving.

Network Wrapper Operation

Data flow within the network wrappers is a fairly straightforward process. The overriding fact is that applications only know about and use the TPipe and TPipeDataUnit interfaces. It is the pipe that deals with the protocol object, configuration, inbound message queueing, and network request transformations.

The scenario that follows illustrates data flow within the network wrappers.

Wrapper Instantiation

As already covered in the document on *The Plumbing*, identities create pipes, and pipes create the pipe data units. That way, there is an explicit binding between the identity, the pipe (a dynamic representation of the identity), and the data unit (the messages recognized by the pipe).

Say, for instance, we have a situation where a client on one machine wants to send a request to a server on another machine. The service interface is transaction-oriented, and the underlying protocol is ATP.

Before the client can send the request, the server needs to set itself up. The first thing the server has to do is get an identity (however that will be done), and ask the identity to create a pipe.

```
/* PB is a reference to the phone book */
TIdentity serverId = PB.GetId(attribute serverList);
TPipe serverPipe = serverId.GetPipe();
```

When the server asks the identity to get a pipe (via the *GetPipe* method), the identity is actually instantiating a pipe behind the scenes. It knows what type of pipe to instantiate based on the type of service requested in the identity. In turn, the pipe temporarily instantiates a configuration object (TConfig) to help it configure itself. The reason the configuration object is a separate class is because many different types of pipes (TNetPipe, TNetStream, etc.) all need to go through the same configuration sequence.

The configuration object does a number of things, among them, choose the appropriate protocol object (TProtocol) to create. It also constructs a protocol path for the network team service interface (TNetService) that forces the network team to set up the appropriate protocols, links, and bind objects. See the following section for more on the network team. Finally, the configuration object instantiates a wrapper task (TWrapperTask) to manage data from the network that has not yet been requested by the user.

Once the configuration is complete, the server must open the pipe and post a receive.

```
serverPipe.Open();
TMessage& request = serverPipe.GetMessage();
serverPipe.ReceiveRequest(request);
```

In this particular case, the *Open* method gets the wrappers to post a listen request to the network team. The *ReceiveRequest* call blocks the server until a request arrives from the network.

The client also needs to create a pipe, where one of the attributes of the pipe is the name and address of the server it wants to talk to.

```
TIdentity clientId = PB.GetId(attribute clientList);
TPipe clientPipe = clientId.MakePipe();
clientPipe.Open();
```

Reading and Writing

Once the client opens the pipe, it can then stream out the request and block on the reply.

```
TMessage& request = clientPipe.GetMsg();
TMessage& reply = clientPipe.GetMsg();
data >>= request;
request.Send(reply);
```

When the client asks the wrappers to instantiate the request and reply messages (via the *GetMsg* method), the wrappers allocate a shared-memory area. The reason for this is that the network team has the ability to read this data without copying the data into its own local address space or relying on a send-with-section from the wrappers.

Once the client sends the request, the message packages its data into a network request and submits itself to the associated pipe. The pipe in turn hands the message to its protocol object so that it has a chance to insert protocol-specific attributes and commands. In this example, the request is synchronous, so the wrappers simply submit the network request to the network team service interface and block on the reply. If the user's send request were asynchronous, however, the wrappers would spawn a task to do the submission.

Eventually, the server gets the request, then replies.

```
request data <<= request;
request.Reset();
reply data >>= request;
```

```
request.Reply();
```

Notice that the server streams the response into the same message that contained the request. The reason for this is that the network request object associated with the request contains information specifically identifying the request. If the server were to stream the reply out to a separate message, this identifying information would be lost, and the network service wouldn't know to which request this reply matched.

Finally, the client can stream out the reply.

```
returned data <<= reply;
```

Network Team Operation

The low-level network operation is accomplished at the client level using two main classes: the network request (TNetworkRequest) for forming requests, and the network team service interface (TNetService) for processing those requests. The first thing that a potential client must do is to create a TNetService object. In order to do this, a TAttributeList must be created with the attributes necessary for creating the protocol path in the network team. Normally, this consists of several attributes: 1) A TTextArrayAttribute with tag *gStringRoute*, and owner *kNoOwner*, which contains a text array of the names of the protocols in the path, from highest layer to link layer; 2) A TTagAttribute with tag *gHardwareLocation*, and the owner code indicating the hardware (i.e. gSccPortB, or gDumbEthernetCard); and 3) A TTagAttribute with tag *gSlotNumber*, or a TTextAttribute with tag *gAroseName*.

This TAttributeList is used to instantiate the TNetService object. If instantiation succeeds (failure will cause an exception - use TRY/EXCEPT/ENTRY to catch the failure), you may begin sending requests.

PROTOCOL	*LISTEN* EQUIVALENT	*SEND* EQUIVALENT
ALAP:	Protocol handler	LAPWrite
DDP:	socket/protocoltype listener	DDPWrite
NBP:	illegal command - can't listen for NBP	illegal command - can't send through NBP
ATP:	ATPGetRequest	ATPSendRequest
PAP workstation:	PAPRead	PAPWrite
PAP Server:	PAPRead	PapWrite
ASP workstation:	GetAttention	ASPCommand/ASPWrite
ASP server:	GetASPRequest	ASPRespond/ASPWriteReply

Table 1: Listen and Send Command Equivalents

COMMAND	DEFINITION
Pause	The protocols stops processing any user request and incoming packets.
PauseClients	The protocols stops processing any user request.
PausePackets	The protocols stops processing any incoming packets.
Resume	The protocols resumes processing user requests and incoming packets.
ResumeClients	The protocols resumes processing user requests.
ResumePackets	The protocols resumes processing incoming packets.
GetRoundTripTime	Handle 'Echo'-type functionality. If the layer does not know how to do this, it passes the request to the next lowest layer. This allows the highest-possible layer to do timing in order to better simulate real operating conditions.
CancelRequest	The protocols cancel that packet with the given id.
ResumePackets	The protocols resumes processing incoming packets.
GetLocalStats	Returns the local statistics for addressed protocols, and sends the request to the next layer for it to process.
GetROConfig	Returns the local configuration for addressed protocols, and sends the request to the next layer for it to process.
OwnerQuery	Each protocol in the path examines the Tags in the original request, and if it is the top-most owner of the tag, it adds its ownership into the reply.
Probe	Each protocol adds its ownership tag into the reply
ResetPacketWindow	Changes the maximum number of outstanding packets that a layer may have at one time.
CancelRequest	Cancel an outstanding request by the 'Timestamp' of it's TNetworkRequest object.

Table 2: Commands Recognized by all Protocols

Listening on a Channel

Normally, the first thing that must be done is to listen on a channel. At least two attributes are required for this command: *channel* and *listener*. If the format of the channel is known, you add the channel attribute and attempt to open the channel. If it is not, you create a channel attribute, which is a template for the channel the Protocol should allocate. In the latter case, the channel will be added to the TDataList for future use. The AppleTalk protocol equivalents for the *Listen* command are illustrated in table 1.

It must be noted that the use of the template channel is not always legal. ALAP, for instance, cannot dynamically create a channel. (Oh, I suppose it could, but currently it's not defined that way!).

Sending and Receiving

The *Send* command can then be used to send a packet (or packets). For connection-oriented protocols, a channel to send on may not need to be specified. The TNetService object may imply the channel. This is true, for instance, for PAP workstation and ASP workstation. It is also not required for ATP (since ATP is 'authorized' to create its own channel), but it is optional. ASP server and PAP server, however, require the channel attribute that was received via the incoming request in order to match the response with the appropriate request. Send requests also require an address to send to, as well as a TMemory object containing the data to send. Table 1 above suggests the *Send* command equivalents for some of the AppleTalk protocols.

The ASPRespond/ASPWriteReply duplicity above can easily be resolved by the channel information from the corresponding incoming request. Unfortunately, the ASP workstation duplicity is not so easily resolved. This is handled by creating two new commands: 'SendRequest' and 'SendWriteRequest' specifically for transaction/connection protocols.

COMMAND	DEFINITION
ReserveChannel:	Prevent a channel from being listened on by another client, but do not accept incoming packets
ReleaseChannel:	Free the channel for others to listen on
Listen:	Receive packets from a specific channel
Send:	Send a packet (channel may or may not need to be specified).
CancelListen:	Cancel a listen request
CancelSend:	Cancel a send request

Table 3: TDatagramLayer commands

COMMAND	DEFINITION
ReserveChannel:	Prevent a channel from being listened on by another client, but do not accept incoming packets
ReleaseChannel:	Free the channel for others to listen on
ListenForRequests:	Receive requests on the specified channel
SendRequest:	Send a request and get the reply
SendReply:	Send a reply to an incoming request
CancelListenForRequests:	Cancel a listen request
CancelSendRequest:	Cancels outstanding request
CancelSendReply:	Cancels outstanding reply

Table 4: TTransactionLayer commands

COMMAND	DEFINITION
RegisterName:	Registers a name.
DeRegisterName:	De-registers a name.
Confirm:	Confirms a name at a specified address/channel, or looks up a single name
Lookup:	Looks up a set of names based on criterion in the TNetworkRequest. In the absence of any other information, the domain your machine is located in is searched. If a list of Domains is included in the dataList, the list of Domains will be searched.
EnumerateDomains:	Returns a set of Domain attributes which are the domains available on the network. For hierarchical domains, you may specify the top level domains, and this will enumerate domains 1 level deeper.
GetConfiguration:	Returns configuration information, including information on whether domains exist, and whether they are hierarchical or flat.
SetConfiguration:	Configures default search domains.
Cancel:	Cancels an outstanding request

Table 5: TNamerLayer commands

Recognized Commands

Protocols are divided into classes which correspond roughly with which commands they understand. The current categories of protocols are: TDatagramLayer (which includes any packet-oriented layers), TTransactionLayer (e.g. ATP), TNamerLayer (e.g. NBP), and TStreamLayer (e.g. ASP, ADSP, TCP).

Table 2 lists the commands that all protocols recognize. The implementation of these commands is mostly handled by the TNetLayer superclass. Table 3 lists additional commands recognized by datagram layer. Table 4 lists additional commands recognized by the transaction layer; table 5, the namer layer; and table 6, the stream layer. Note that the TStreamLayer classification includes commands for transaction-oriented as well as data-oriented streams.

Deliverables

The BabelFish project plans to ship the entire AppleTalk stack with the first release of Pink. We will allow the AppleTalk stack to be multi-homed. The protocols included in this are:

- LocalTalk, EtherTalk, and TokenTalk
- DDP
- ATP
- ADSP and ASDSP (secure ADSP)
- ASP

Services included in BabelFish include wrapper integration with the message and pipe architecture. We also plan on shipping an additional network toolbox to better support clients such as AppleShare and collaboration.

COMMAND	DEFINITION
CreateConnection:	Establishes a connection with a specified address (and an optionally-specified channel).
ListenForConnection:	Listen for a connection request
RefuseConnection:	Refuses a connection request
AcceptConnection:	Accepts a connection request
CloseConnection:	Closes the specified connection
Listen:	Listen for incoming data on a connection
ListenForRequests:	Listen for incoming requests
SendReply:	Send a reply to a read request
SendRequest:	Send a request which will cause data to be sent to us.
Send:	Sends data on the main connection on side-channel to the client
CancelDataList:	Cancel an outstanding datalist
CancelListen:	Cancel a listen request
CancelListenForRequests	Cancel a listen for requests
CancelListenForConnection:	Cancel a listen for connection

Table 6: TStreamLayer commands

One our wish list, we hope to implement TCP/IP, time permitting.

Projects Associated With BabelFish

The projects associated with BabelFish are:

- *Phone Book (Pathfinder)* - We needed a way to create, locate, update, authenticate, and search for identities. Think of *Pathfinder* as replacing the machinery behind the Chooser in Pink.

- *Messages and Pipes (The Plumbing) - The Plumbing* defines the wrappers' interface to the user of the network. Messages and pipes are absolutely necessary if the network is to fit into Pink seamlessly.

- *Collaboration on Pink* - One of our stated goals from the beginning was to make collaboration a default behavior in Pink. To this end, we are extremely interested in collaboration and intend to implement some of the ideas in ShareBoard on Pink. This will hopefully help drive the design of services built around networking.

- *AppleShare* - AppleShare is one of our biggest clients, and just as with collaboration, we hope AppleShare will help drive the design of network service too.

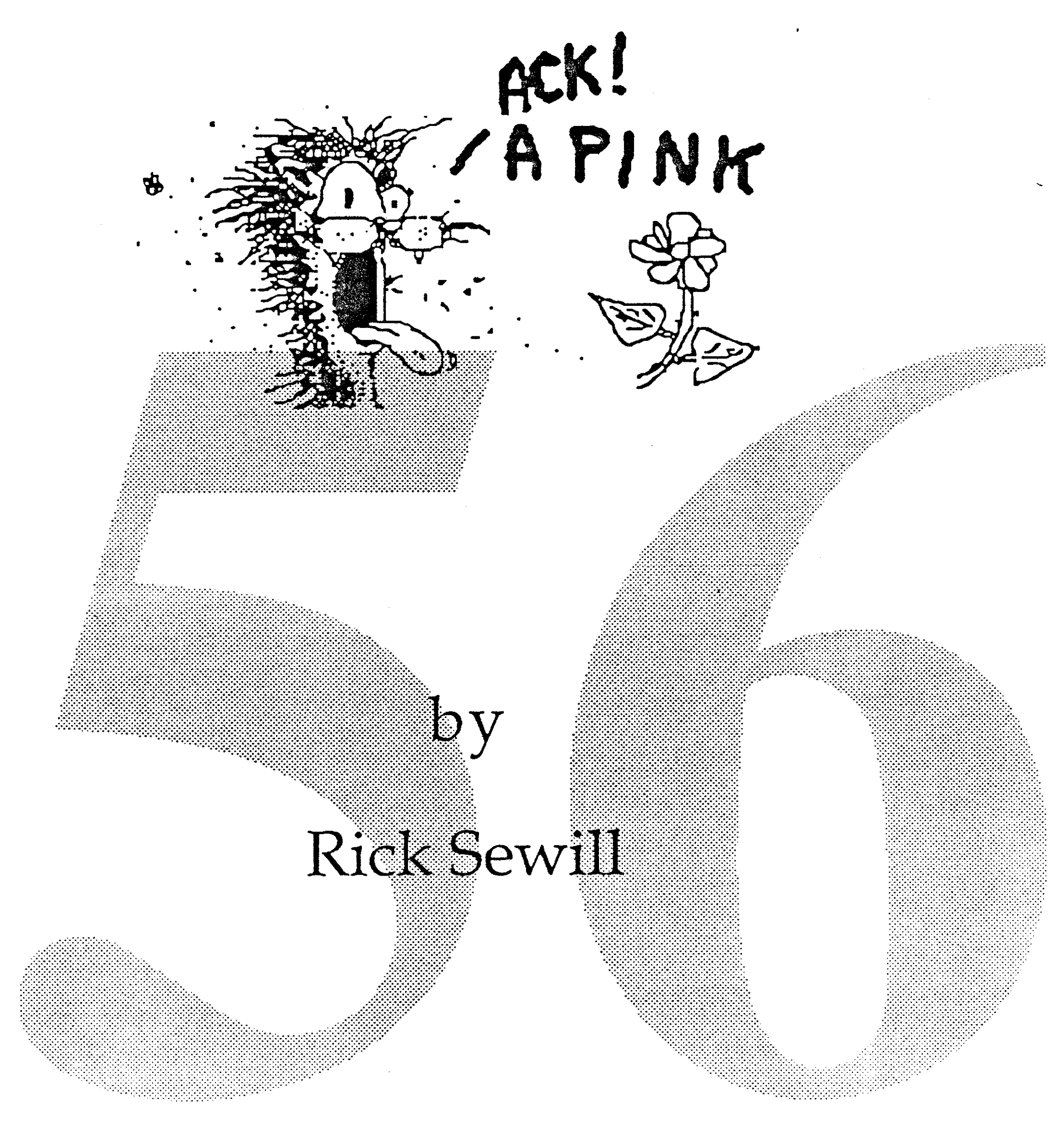
- *Network Blue Adapter (Scorpion)* - *Scorpion* will certainly exercise the network team's interface and provide valuable input to the continued maintenance of the protocol and utility classes.

- *A-Rose on Pink (Coral Dawn)* - Here again, a valuable client of the network team.

Coral Dawn

A/ROSE

56



by

Rick Sewill

56

A/ROSE Is

A/ROSE is a real-time kernel running on 68000-based smart NuBus cards. A/ROSE is a small kernel with task scheduling. A/ROSE is a fast kernel with fixed length IPC messages for delivering events. A/ROSE is a helpful kernel with procedures for moving data across the NuBus. A/ROSE is a friendly kernel with memory management. A/ROSE is a collection of user tasks called managers. These managers provide high level services including naming, inter-card communication, and dynamic task downloading support.

A/ROSE is an access manager on the main logic board. The access manager provides a subset of services found in the A/ROSE real-time kernel. Opus tasks use the access manager services to work with A/ROSE tasks running on smart NuBus cards.

A/ROSE is a set of interface classes between Opus tasks and the A/ROSE access manager.

A/ROSE is a set of support classes for downloading tasks to smart NuBus cards.

A Real-Time Kernel

A/ROSE is a real-time kernel running on smart NuBus cards. The *MCP Developer's Guide* describes the A/ROSE kernel in detail.

I do not have any plans to change the code running on the NuBus cards at the time of this writing. It might be interesting to try and implement a subset of the Opus kernel wrapper classes as a library within A/ROSE and a support manager running under Opus.

An Access Manager

A/ROSE is an access manager on the main logic board.

The access manager installs an ISR for receiving NMRQ interrupts from smart NuBus cards.

The access manager sends A/ROSE fixed length IPC messages to the smart NuBus cards running A/ROSE. A/ROSE forwards these messages to the user tasks.

The access manager receives A/ROSE fixed length IPC messages from smart NuBus cards running A/ROSE. The access manager forwards these messages to Opus tasks.

The access manager moves data across the NuBus on behalf of tasks on the main logic board.

Interface Classes

A/ROSE is a set of interface classes between Opus tasks and the A/ROSE access manager.

A/ROSE IPC messages can be created using the interface classes; A/ROSE IPC messages can be deleted. A/ROSE IPC messages can be sent using the interface classes; A/ROSE IPC messages can be received. These IPC messages are for event delivery.

The interface classes have methods for moving data across the NuBus. Yes, yes, I know. Only tasks within the same team as the A/ROSE access manager can actually move data across the NuBus. The interface classes really get the data to the access manager and request that the data be moved.

Support Classes for Downloading Tasks

A/ROSE is a set of support classes for downloading tasks to smart NuBus cards.

The support classes find out what cards are in what NuBus slots.

The support classes determine if A/ROSE is running on the smart NuBus card in a given slot.

The support classes download A/ROSE to a smart NuBus card.

The support classes determine if a user written task is running on a smart NuBus card.

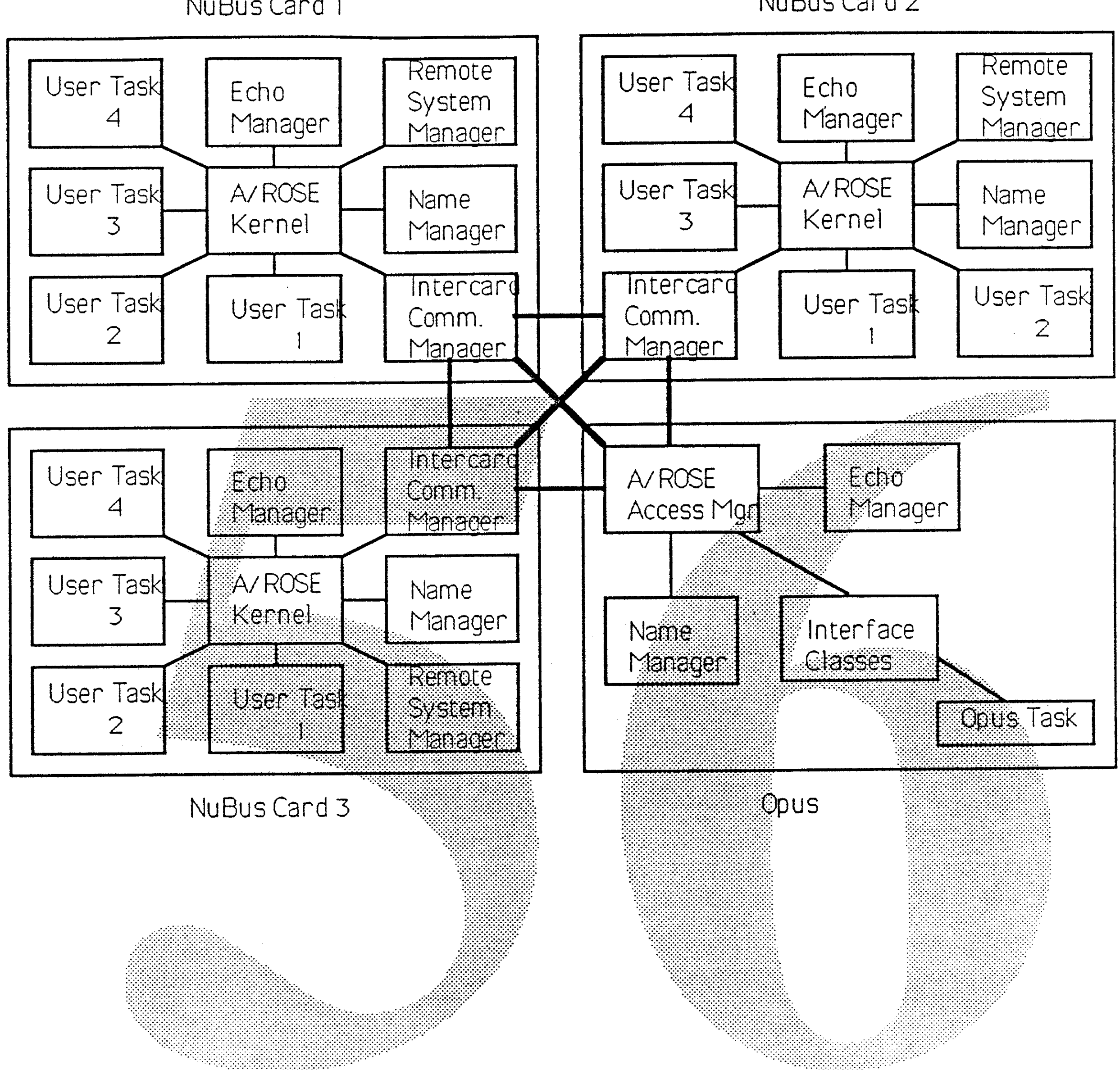
The support classes download user written tasks to smart NuBus cards.

Now can anyone tell me what A/ROSE is?

I bet you can!

A/ROSE Look and Feel

56

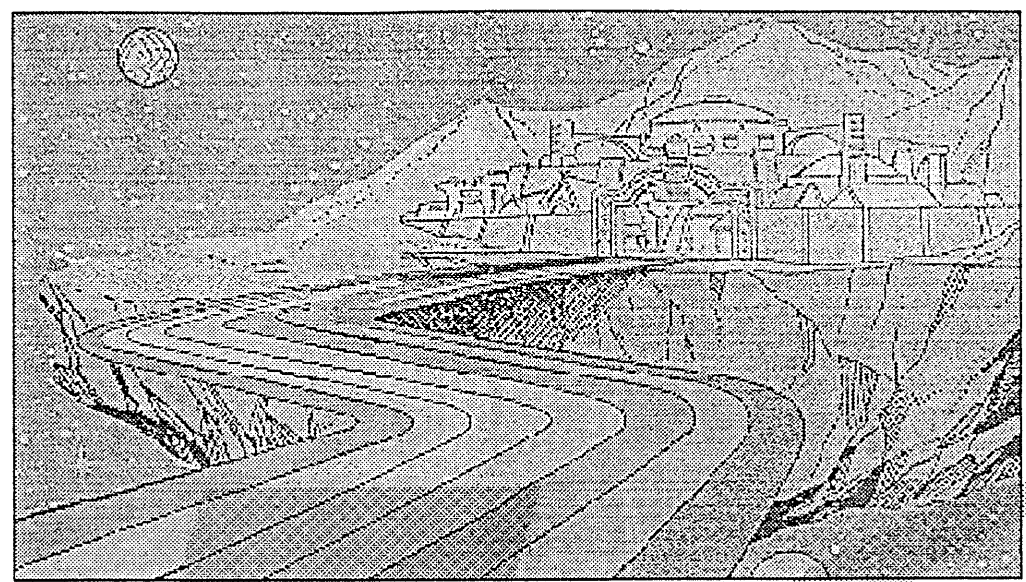

NuBus Card 1
User Task 4
Echo Manager
Remote System Manager
User Task 3
A/ROSE Kernel
Name Manager
User Task 2
User Task 1
Intercard Comm. Manager
NuBus Card 2
User Task 4
Echo Manager
Remote System Manager
User Task 3
A/ROSE Kernel
Name Manager
Intercard Comm. Manager
User Task 1
User Task 2
User Task 4
Echo Manager
Intercard Comm. Manager
User Task 3
A/ROSE Kernel
Name Manager
User Task 2
User Task 1
Remote System Manager
NuBus Card 3
A/ROSE Access Mgr
Echo Manager
Name Manager
Interface Classes
Opus Task
Opus

56

Valhalla

The Pink Finder

56



Valhalla

Valhalla provides the tools people use to manage their Pink systems and the information on them. The overall goal of Valhalla is to make Pink more personal, easier to learn, easier to use and more effective to use than any other commercially available system. Valhalla offers power while keeping the Pink system simple to understand and use. It is extensible and designed to grow with the user's needs throughout the life of the system.

Valhalla takes advantage of Pink features such as multitasking, hypermedia linking, content-based retrieval and shared libraries to deliver specific benefits to its two audiences. For end users, these benefits include quick document finding and no more time wasted being locked out of the system waiting for lengthy operations to complete. For developers, this includes being given a simple and consistent way to display and manipulate desktop objects, as well as the ability to move desktop objects between applications in a straightforward and consistent manner.

Joel Spiegel
Steve Horowitz
George Norman
Karl B. Young

56

Contents

About this Document .... page 4

Valhalla Project Mission .... page 5

Introduction .... page 6

Thor Product Description .... page 8

Odin Product Description .... page 15

56

About this document

This document provides a description of the current state of the Valhalla project. It is meant for anyone interested in the direction and current status of Valhalla. Readers should be familiar with the Pink project in general. They should also be familiar with the history of the Macintosh and Macintosh Finders.

This is a working document. It describes what is currently known about the project. It will be updated every four to six weeks.

What is in this document?

- **The Valhalla project mission**, briefly describes the motivation for the Valhalla project.
- **Introduction**, describes Valhalla as a whole and introduces the specific components of the Valhalla project.
- **Thor product description**, provides a high level overview of Thor, the user interface component of Valhalla.
- **Odin product description**, provides a general description of Odin, the shared class library component of Valhalla.

Related documents.

There are two other documents which describe various aspects of Valhalla. These are described below.

- *Thor User Interface Specification*, describes the Thor user interface. This includes a description of objects on the desktop and how they are manipulated by the user. This document is of interest to anyone who wants to understand the user view of Valhalla in detail.
- *Odin Programming Interface Specification*, describes the desktop services available to application developers (including the Thor team). These are the programming interfaces used to display and edit desktop objects. This document is of interest to anyone who wishes to write an application which manipulates objects on the desktop. Under Pink, this includes all applications which wish to create or modify documents.

The Valhalla project mission

Valhalla will translate Pink core technologies into concrete end user and developer benefits. The result will be a personal computer environment which is perceived as easier to learn, easier to use, and more effective to use than any other system available.

The mission of the Valhalla project is to make Pink the most compelling and productive environment available to personal computer users in the 1990's.

Introduction

Valhalla is the Pink equivalent of the Macintosh Finder. It provides the means by which people control and interact with their Pink systems. Like the Macintosh Finder, Valhalla supports an interface which is strongly rooted in real world metaphors and is visually oriented. Valhalla is actually made up of two main parts, Thor and Odin.

Thor

Thor is a set of user interface programs which provide end users direct access to objects in the metaphorical world. It is the Pink analog of the Macintosh Finder. Thor allows people to control and use their systems and the information on them.

Odin

Odin is a set of classes which support the Desktop metaphor. This is similar to, but broader than, the desktop metaphor. They support a model of the world which embodies all of the things which can exist and all of the operations on these things. This model is extensible so that new types of things can be added to it. Odin also provides an initial set of user interface presentation and editing capabilities. These will be used by Thor. They can also be used by any other application wishing to operate in the Valhalla environment.

The common purpose of Thor and Odin

The purpose of both Thor and Odin is to handle two key aspects of the user's world:

- **System management** is the management of the system's hardware and software. It includes basic configuration of the system. It also includes modifying the basic behaviors of the system to meet personal preferences. Examples are installing applications and setting preferences on the Macintosh. Basic troubleshooting also falls into this domain.

- **Information management** consists principally of organizing and finding information on the system. On Pink, this information is contained in documents. A less obvious, but significant job is to provide access to the programs which let users edit documents.

Thor and Odin are partners in these functions. Odin provides the underlying machinery for them. Thor delivers user level control. It is an explicit goal that the functionality exported via Thor map directly to core functions of Odin.

Anyone who uses a personal computer must deal with these two issues. The degree to which they are made transparent, or at least simple, has a tremendous impact on people's perceptions of the machine. The simplicity of performing these tasks is a key criterion which must be met in order for Valhalla to be considered cost effective and easy to use.

The strategic significance of Valhalla

Today's Macintosh Finders present the friendliest and most productive personal computing environments available today. Despite the success of our current Finders, two situations make it

imperative that we develop a replacement for today's Finders. These are:

- **New system software on classic Macintosh hardware.** Pink will deliver, through software, a number of new technologies to our users. This will alter the entire OS and toolbox interface on the Macintosh. Porting NewFinder to a system with radically different underlying semantics is not reasonable. We must present the new Pink capabilities in a coherent manner in the Pink user interface.

- **Next generation hardware platforms.** Jaguar will use Pink as its system software. Its user interface shell must run effectively in the Pink environment. In addition Jaguar will provide hardware support for new features such as multimedia input and presentation. This requires a user interface framework which can integrate these concepts effectively.

Valhalla represents the next generation of Finder technology at Apple. It is a significant evolutionary step. Like its predecessors, it is based on a set of real world metaphors. Users can have a concrete and understandable world to work in.

Unlike its predecessors, Valhalla takes the metaphors and extends them beyond the desktop. Furthermore, it is being designed with a number of new technologies in mind. This includes software technologies such as preemptive multitasking and hypertext linking. It also includes hardware technologies such as stereo sound and video input and output.

Valhalla will translate these technologies into a broad range of specific benefits for our customers. Some examples include more effective finding of documents by attribute and content, as well as more effective use of time through background processing of compute intensive tasks. By providing the right benefits, Valhalla will help our customer's perceive Pink as being easy to learn, easy to use and effective to use.

What Valhalla is not

It is important to understand that Valhalla is not meant to solve domain specific problems. It provides general services. Although it can help find a specific invoice or drawing,there is no way to edit accounting or architectural information. Other programs, usually applications, are required for this.

Thor Product Description

Thor is a set of programs which create the user environment on Pink systems. Thor presents a consistent user model based on the Desktop metaphor. It allows people to manage their systems and the information on them.

Who are Thor's customers?

This is really the same as asking who are Pink's customers? Anyone who uses a standard Pink system will use Thor. This includes end users in business and education. It also includes developers.
This is an extremely broad subject. It is covered in more detail in other Pink user interface documents as well as the Thor User Interface Specification.

For all practical purposes, we can consider our potential customer base as any normal person who needs to use a computer.

Key characteristics

Thor is designed with four key characteristics in mind. These can be thought of as high level, or general benefits.

- **A truly personal computer environment.** A personal computer reflects the needs of its user. It is fully under the user's control. Since people often work with other people, this need must be accommodated by Thor. Although Thor will facilitate work in group environments, it does not dictate how people should interact.
- **Easy to learn.** Normal people will find Thor easy to learn. Most people will be able to learn to use Thor effectively without having to learn any new or arcane concepts.
- **Easy to use.** Once a person has learned the basics of Thor, Thor is easy to use. This is accomplished by providing ample and timely feedback as well as keeping the number of interaction concepts small. A key component of this is relying on recognition as opposed to recall.
- **Effective to use.** Despite being simple to use, Thor is not limited to doing simple things. It will be straightforward to specify sequences of operations as well as operations on groups of objects.

Each of these is regarded as a desirable attribute for a PC. We know of no system available today which is perceived as having all four of these characteristics. Although the Macintosh comes close, it is currently considered weak in terms of the "effective to use" characteristic.

What specific benefits will Thor provide?

From a user's point of view, this is like asking what are the benefits of Pink? Thus, we present a list of key perceived benefits, whether or not they are unique to Thor. Many, in fact, are derived directly from simply existing in the Pink environment. Also ,some of these benefits are already found to some extent in the Macintosh Finder. These are still listed as key benefits of Thor either because they are critical to the overall user experience, or because Thor takes them significantly further than the Finder.

Easy to set up a Pink system.

This is the first problem faced by the owner of any computer system. Since Thor is our standard interface, it should handle this task. If Pink systems come preinstalled, this task will be simplified, but not eliminated. Users will be able to add and configure new hardware easily within the standard framework.

Comparison with current Macintosh:

WIth our new installer, system installation on current Macintosh systems is fairly straightforward. The greatest shortcoming of the current scheme is that it requires learning a program, the Installer, which is distinct from the world users normally operate in. This will not be the case with Thor.

It is easy to install additional software.

Software installation will also be done through standard means. There is no need for arcane installation procedures. Thor will provide facilities for all needed software installation mechanisms within the normal desktop framework.

Comparison with current Macintosh:

Installing some packages involves running special programs such as installers or Font/DA Mover. Some also require users to be aware of the special nature of the System Folder. The new installer has simplified things somewhat. Thor will simplify this process even more by reducing all installations to simple copies. While there may be some magic going on behind certain copies, this should be transparent to the user.

Easy to organize things.

Organizing things is done by means of Folders or other well known types of containers. As with any metaphor based system, an object is concrete and exists in only one place at a time.

Comparison with current Macintosh:

Thor's basic mechanisms for organization are exactly the same as with Finder and NewFinder.

Easy to find things.

Thor provides a variety of ways to find things. These include manual searching, as well as searching by attribute, by association and by content. Users may select those most appropriate method for the task at hand.

Comparison with current Macintosh:

Our original Finder supported only manual searching. Other techniques are added by things like Find File and ON Location. NewFinder adds the ability to find by attribute and by association. Point by point, here are the key differences between NewFinder and Thor.

- Content - Thor will be the the first Apple environment to present this ability to users in an integrated manner. Users can, using normal Thor techniques, find documents containing specific words or objects anywhere on the Desktop.
- Association - Thor will handle this using a concept called Cross-references. These are similar to NewFinder Aliases. Unlike Aliases, Cross-references can be bidirectional and of varying strengths. In addition, the Thor concept of association will be integrated with the document level hypertext linking mechanisms in Pink.
- Attribute - Thor does this much like NewFinder. Thor is, however, somewhat more flexible than NewFinder.
- Manual searching - Thor does this just like Finder and NewFinder.

Lets you view the world as if it were organized in different ways.

Thor will allow you to create different views of the world based on what you are interested in. You will be able to specify the sorts of things you want, such as "all 1989 documents dealing with dolphins and fish". Thor will create a list (either iconic or textual) which matches your specification. This lets you view the world as if it were organized from different perspectives.

Comparison with current Macintosh:

Although this is not impossible on Blue, we are not aware of any plans to incorporate this ability. John Thompson-Rohrlich of ATG is currently doing work on this subject.

Reduces mistakes.

Many computer users make mistakes which should be easily avoided. Thor reduces this problem by giving people adequate information about the state of the world. This includes relying on recognition rather than recall. Timely feedback about the state of specific objects also helps to reduce errors.

Comparison with current Macintosh:

Compared to most other systems, Macintosh does very well in this area. Thor takes things yet a step further. Consider shared resources on a network. Thor always shows everything relevant which can be determined about a given object. For example, if you are looking at a document in a window and someone else is using it, this fact is made obvious by the state of the icon. Thor will also support animated icons in order to provide additional information about the state of the system.

Reduces fear and intimidation - increases fun:

Fear of damaging the system or of losing data is a great source of stress for many users. Thor presents a very safe world to our users. Requested operations either succeed or abort gracefully. The world is never left in an inconsistent state. Furthermore, as many operations as possible are trivially undone. This opens the door to stress free exploration of the system.

People do not enjoy looking up solutions to their problems in manuals. Many will often suffer a great deal before picking up a manual. Thor provides easy, online, help at all times.

Comparison with current Macintosh:

Once again. Today's Macintosh does a very fine job. Compared to other systems currently available, it is low stress and high fun to use. There is, however, room for improvement. Thor adds multi level undo. Transaction based operations are also new to Thor.

Apple's Human Interface Group has developed promising designs for several kinds of help. NewFinder will implement only "what is" help. Thor, as a standard Pink application, will implement at least "what is" and "how do I" help.

No wasted time due to unnecessary waiting(ban the watch):

On many systems, users must wait for a specified operation to finish before they can do anything else. Thor rectifies this problem. It always returns control of the system to the user as soon as possible. Whenever possible, time consuming operations are carried out by the system while the user is free to carry out other tasks. The only time a user must be concerned with a previously requested operation is if they want to do something which conflicts with that operation.

Comparison with current Macintosh:

Macintosh, even under 7.0/NewFinder is very limited in its multitasking abilities. Thor will provide this benefit consistently in the standard user interface.

Reduces needless repetition:

People often perform the same operation or sequence of operations many times. This is generally very tedious. Thor allows you to generate scripts which represent sequences of commands.

Comparison with current Macintosh:

This sort of feature is not presently available on Macintosh in any sort of unified manner. MacroMaker provides some help. It is generally considered very inadequate. Hypertalk provides this benefit within the HyperCard environment. In addition, there is evidence that Hypertalk is principally used only by people who can be reasonably described as programmers. Thor, like all other Pink applications, provides this sort of scripting in a standard, consistent and (we hope) uniquely useful manner.

Specify operations on arbitrary groups of objects:

Although you will often want to perform an operation on groups of things in the same place, there are times when you may wish to use other criteria. For example, you may wish to copy all of your letters on a particular subject to a floppy disk, regardless of where they exist in the system. Thor will allow you to do this.

Comparison with current Macintosh:

This ability is not currently available on Macintosh. Although it should be possible (within limits) under NewFinder, we do not expect it to be available when 7.0 is released.

A customizable environment:

Many people with to customize their systems in many ways. Sometimes this is simply a matter of personal preference. Sometimes it is necessary due to physical handicaps. Thor is designed to allow many details of the interface be modified, supplemented or even completely replaced. Increasing or decreasing the significance of any specific media such as sound or graphical display is easily accomplished.

Comparison with current Macintosh:

To our knowledge, Thor accomplishes this to a unique degree.

Natural environment for sharing:

The ability to share computer based resources with coworkers is usually not a natural part of most systems. Thor will allow sharing of all desktop objects in a simple and effective manner. This includes printers, documents, folders, etc.

Comparison with current Macintosh:

Personal Appleshare (aka Killer Rabbit) is a great step toward this goal. It is, however, still an add-on to the file system and the system as a whole. Thor will do for all system components what Personal Appleshare does for files. In addition, Thor will provide a consistent system-wide mechanism for controlling access to your system's resources.

Allows work with more kinds of information:

Thor's extensibility allows you to work with the kinds of information natural to your environment. Thor can support animation or even real-time video to show the state of objects in the system.

Comparison with current Macintosh:

To our knowledge, this facility is unique to Thor.

What features will help provide these benefits?

This section discusses the key Pink technology features which will help us to deliver end user benefits. For each feature, we describe what the feature is, where the feature is derived from and what key benefits it helps provide.

Extensive use of metaphor

Metaphors allow people to formulate useful mental models of the system more quickly. The Thor user interface is firmly rooted in a consistent set of commonly known metaphors. Objects visible to users all make sense in this metaphorical world.

The metaphors made visible through Thor are actually implemented in the Odin libraries.

Use of metaphor helps make Thor easy to understand. This helps with benefits related to ease of learning and ease of use. It makes the system understandable enough that set up and configuration should not be intimidating.

Iconic interface

Thor uses icons to represent objects in the metaphorical world. Icons help make the metaphorical world concrete. They are responsible for providing users with a concise and accurate view of the work. Thor icons can incorporate animation, video, etc to to accomplish their jobs.

The underlying mechanisms for implementing iconic user interface elements are provided by the Odin class libraries. In addition, Odin provides specific iconic views of the basic objects in the system. These are also used by Thor.

With even the most basic understanding of the underlying metaphors, icons can help people find and organize information. As a user's knowledge grows, the icons can convey tremendous information about the state of the system. By providing timely and accurate information about the state of the system, the icons help reduce uncertainty and intimidation.

Direct manipulation

Users explore and change their world by directly manipulating the icons which represent specific objects. These user level manipulations of icons are translated one to one into operations on the objects themselves.

Once again, Thor makes heavy use of the Odin libraries to accomplish this. The user interface elements of Odin already know how to translate manipulations into user specified operations.

This feature also helps people move into the Pink world quickly and easily. There is no translation from abstract commands into operations. The results of operations on icons are intuitive and directly visible. This reduces significantly user intimidation.

Fast/orthogonal context switch

When a user's interest shifts, Thor must accommodate this as quickly as possible. When a user clicks in a new window, Thor changes its focus to match what the user wants. This may involve selecting a new icon or bringing a specific window to the front. Only those things the user has indicated an interest in are affected.

There is no concept of application layers, etc. A shift of focus does not imply alot of baggage. Under the model of direct manipulation, only the thing being manipulated and those things naturally attached to it should be affected.

This feature is principally inherited directly from various parts of the Pink application framework.

This feature helps reduce confusion and stress. It also reinforces the feeling of direct manipulation <<<is it really part of direct manipulation?>>>

Multitasking

From a user perspective, this is the ability to have more than one thing actually happening at once. Simple examples include background printing and copying.

The principle benefit Thor derives from using Pink's multitasking abilities is that it can always return control of the system to the user as quickly as possible.

Thor derives its multitasking features both directly from the Pink application environment and by using the Odin libraries which are intrinsically multitasking.

Fast feedback

When the world can change, or when it does change, Thor makes this immediately visible to the user by changing the state of icons. This feature is derived from Pink's multitasking model as well as the services provided by the Odin libraries. This feedback is needed to reduce fear and intimidation. It also helps users learn by exploring.

Online help

Online help is always immediately available. This is a function of being a Pink application. Once again, this feature reduces fear and stress, and encourages learning.

Scripting

Scrips represent sequences of operations. These operations can be done again and again simply by invoking the scripts. Scripting is an intrinsic part of all programs derived from the Pink application framework. The principle benefit of scripting is that it reduces needless repetition. It can also reduce easily avoided errors by automating complex sequences of commands.

Hyper-media links

In the Thor context, these represent user specified connections or links between any two things of interest. The basic technology for managing hypertext links is directly derived from CHER. Hyper-media links are represented in Thor as Cross-references. They are used in finding and organizing.

Content based retrieval

This is the ability to find objects based on their contents. This ability is principally derived from CHER. Some aspects may also be implemented in the Odin libraries. This feature is principally of benefit in the area of finding and organizing.

Extensible

Thor support s the ability to add new kinds of views for existing objects. Furthermore, it even allows the addition of new kinds of objects. This feature is derived from use of the Odin libraries as well as the general shared library feature of Pink. Thor's extensibility allows user's to customize their machines.

Support for alternative presentations

Users can run other programs which present alternative or supplemental views of the metaphorical world. Thor's operation is not disrupted by this. Thor will continue to reflect accurately the state of the world when alternative means are used to manipulate objects. This is directly derived from using the Odin libraries. This feature is an aid in letting users customize theirs systems.

Network compatible

Thor desktop objects can be shared over the network. In addition, Thor provides browsing abilities on the network. This is a function of Pink's fundamental networking abilities plus its general client/server model. Network compatibility helps users share information using Thor.

Odin Product Description

Odin is part of Valhalla. It is a set of class libraries written in C++. The classes which make up Odin fall into two groups. These are:

- **Desktop classes.** These classes represent the desktop objects on the system. They provide a complete model of the desktop and the things you can do on it.
- **Presentation classes.** These classes allow users to manage the desktop world through direct manipulation. They provide operations for displaying and managing desktop objects .

Who are Odin's customers?

Odin is intended for use by Pink application writers. Since Odin is responsible for the creation and management of documents as desktop objects, all application writers will use Odin to some degree. Application writers who wish to write the Pink equivalent of Finder extensions will use Odin extensively.

What benefits will Odin provide?

Odin provides a number of benefits to developers. It provides classes which implement all of the functionality most developers will ever need from the desktop world. In addition, it provides a straightforward framework for adding or modifying specific behaviors.

Standard look and feel is available "free".

The Odin libraries support all of the basic user interface operations used in Thor, the Pink User Interface Shell. Both the underlying desktop object classes and high level user interfaces are available to all developers. In simple cases, writing the user interface for Pink "Finder Extensions" involves simply using the Odin classes.

Comparison with current Macintosh:

NewFinder supports a notion of Finder extensions. Finder extensions are a special class of program. Extension authors have the ability to blend in as part of the NewFinder interface. Odin lets any application author do this simply instantiating appropriate Odin classes. These are architecturally compatible with the rest of Pink.

Alternate user interfaces are easily supported.

All of the semantics of desktop objects are separate from the user interface classes. You can extend these easily to produce entirely new, yet compatible, presentations of the desktop world. These presentations can be graphical views or they may even be acoustic presentations. You can use your new classes to supplement the standard interface or to provide alternative views of the world.

Comparison with current Macintosh:

The interface and semantics of the Blue/NewFinder desktop world are very tightly coupled. Odin makes a clean split between them. The standard Odin user interface components can be completely replaced. In addition, supplemental interface classes can be added. The basic desktop services are unaffected by this. NewFinder does not support this ability.

The desktop is always consistent.

Odin is responsible for the consistency of the desktop world. Most of the overhead of coordinating access to the desktop is handled by Odin.

Comparison with current Macintosh:

Blue, even under NewFinder is principally a file based world. Odin pushes the user metaphor down to the programming world.

Developers can extend the desktop world.

New types of desktop objects can be added to the system. Odin provides a framework for adding both objects and operations on objects to the system. Refinements of existing classes are easily added.

Comparison with current Macintosh:

This is possible under NewFinder. Odin takes things a step further by providing full management services for these objects.

What features will help provide these benefits?

This section discusses the key Pink technology features which will help us to deliver Odin's developer benefits. For each feature, we describe what the feature is, where the feature is derived from and what key benefits it helps provide.

Object-oriented framework.

All Odin services are provided by well defined classes. All of Pink is based on an object oriented architecture. Odin fits in with this architecture. The object oriented design of Odin provides a clean model for understanding, using and extending both the desktop and user interface classes.

Shared libraries.

Shared libraries allow one program to take advantage of classes introduced by another program. Shared libraries are a basic part of Pink. Shared libraries provide the basis for extending the desktop world. They are also used for allowing arbitrary applications to use both the desktop and user interface classes.

Client/Server model.

This allows, among other things, one team to coordinate and service requests from a number of other teams. This is a standard part of the Pink toolbox architecture. The client/server model provides the basis for implementing the Odin Desktop services. This provides a framework for guaranteeing consistency of the world model.

Concurrency control and recovery

In a nutshell, this guarantees that desktop operations appear to happen in a sensible order and that each either succeeds or aborts with no visible side effects. This is achieved through user of the Pink Credence classes. This feature is crucial to guaranteeing consistency of the Odin desktop world.

Change notification for clients.

Odin will notify clients of important changes in the desktop model. This is accomplished via Pink IPC. This mechanism eliminates the need for clients to always poll to find out the state of desktop objects.

56

User Interface Specification

Joel Spiegel
Steve Horowitz
Karl B. Young
George Norman

56

Contents

About this Document .... page 3

Introduction .... page 4

The Thor Desktop Metaphor .... page 5

Things in the Desktop World

- Disks & Disk Drives .... page 6
- Folders .... page 8
- Cross-References .... page 9
- Documents .... page 10
- Stationery .... page 12
- Applications .... page 13
- Appliances .... page 14
- Tools .... page 15
- Components .... page 16

56

About this document

This document is intended for anyone interested in the user interface of the Pink version of the Finder (Valhalla). It will also be of interest to people who want to know how Pink in general will deal with aspects of system configuration and organization. Readers should be familiar with the Pink project in general. They should also be familiar with the user interface presented by current Macintosh Finders.

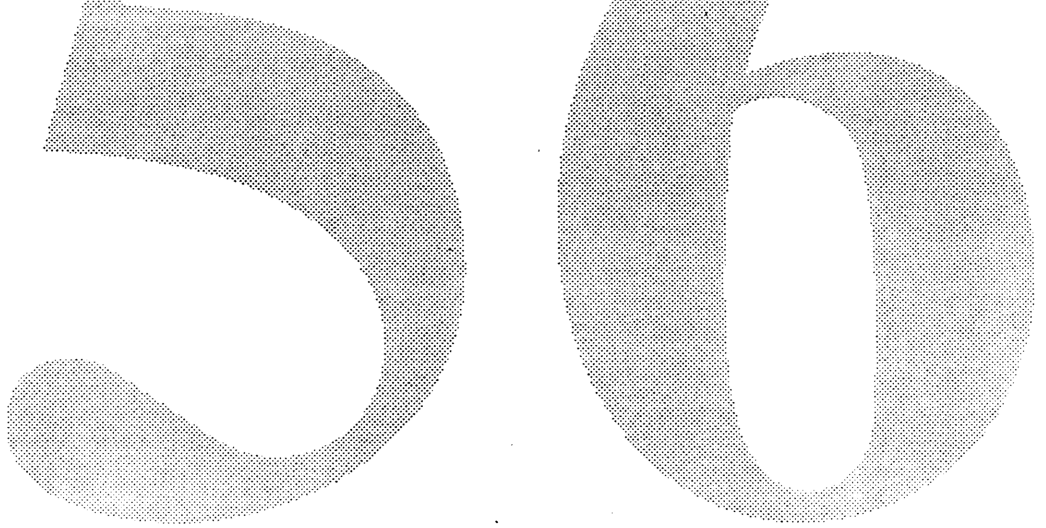
This is a working document. It describes what is currently known about the project. It will be updated as needed.

Related documents

There are two other documents which describe various aspects of Valhalla. These are described below.

- *Valhalla Project Overview*
  Describes the Valhalla project as a whole. Valhalla is an umbrella project which is made up of the Thor user level applications and the Odin desktop services.

- *Odin Programming Interface Specification*
  Describes the desktop services available to application developers (including the Thor team). These are the programming interfaces used to manipulate and display desktop objects. This document is of interest to anyone who wishes to write an application which manipulates desktop objects. Under Pink, this includes all applications which wish to create or modify documents.



Introduction

Thor is the collective name for all applications within the Valhalla project which deal with aspects of user interface for the Pink Finder. These programs present the system and the things it contains in an easy to grasp manner. Using Thor, you can manage your system and the information on it. Thor also allows you to easily install and configure software and hardware as well as organize and find information on your system. Thor is the Pink equivalent of the Finder and its associated desk accessories and utilities on the Macintosh.

Thor is built upon the services provided by Odin (see related documents). Thor uses these services for both manipulation and display of desktop objects.

Components of Thor

The fundamental component of Thor is the Mjolnir application. This application is responsible for the overall presentation and manipulation of the desktop objects to be described later. Other components of Thor include such things as the Macintosh tool, the trash can, and any other user interface application which will ship as part of the Valhalla project.

Intended Audience

Thor, in its standard form, is intended for people with average intelligence, elementary school education level, reasonable corrected vision and normal hand/eye coordination. No previous computer experience is needed to use Thor effectively. Thor is, however, explicitly designed to allow parts to be replaced or supplemented in order to accommodate people with physical handicaps, special needs, or any other personal preferences.

Thor and the Macintosh Finder

Thor is the Pink version of the Macintosh Finder. It has the same basic functions and responsibilities. From a user's point of view, these responsibilities are carried out in a very similar manner. Anyone familiar with Finder should be able to work in the Thor environment immediately.

Thor's user interface is based on a metaphor which is compatible with the Macintosh desktop metaphor. Many of the objects in the Thor world are, for all practical purposes the same as their counterparts on Macintosh. Like the Macintosh Finder, Thor displays objects in the world as icons in windows. Also, like its predecessor, Thor relies more on recognition than recall. However, Thor is distinct from the Macintosh Finder in that it will add some new functionality such as more powerful searching capabilities as well as correct some deficiencies such as the need to sit and wait while operations complete.

Thor Desktop Metaphor

Overview

Thor is designed to let normal people take advantage of the power of modern computers without having to become computer experts. We accomplish this by relying on the common knowledge we share about things in the real world. The world presented to users by Thor is that of an artificial desktop world. This desktop world attempts to present objects and behaviors such as those you would find in an office or on a desktop. By using the familiar metaphor of a desktop, users of Thor can easily manipulate the things they find on their computer as they would items in the real world.

The desktop world contains things which we refer to as objects. Each object has a set of attributes which describe the object. In addition, each object has a set of operations you can perform on it. These have been crafted with two goals in mind. First, each has an important role to play in making your system useful. Second, each bears enough resemblance to some real world counterpart that you should find it reasonably understandable.

Thor will be designed to follow the desktop metaphor as literally as possible. For example, in the current Macintosh there is one object which encompasses all of the functionality of both floppy disks and disk drives. This object appears on the desktop as a single floppy disk. In Thor we will separate the functionality of these two objects and thus both a floppy disk drive as well as a floppy disk will appear on the desktop. By adhering to a more literal metaphor Thor will gain both added functionality as well as be easier to comprehend by users.

Objects as Reflection of Reality

Objects in the desktop world have many of the same properties as objects in the real world. Each is a real thing and can be manipulated, grabbed or moved just like an object in the real world. Thor objects also obey certain laws. Since seeing is believing, if you can see it, it is real. Being real, it exists in one and only one place. Like real objects, Thor objects all have attributes by which you can describe them. In addition, some are containers of other objects.

Desktop Object Attributes

Each desktop object has a set of attributes. These help us to describe meaningful things about the object. Examples include:

- Size
- Creation date
- Who created it
- What its container is
- Where it is in its container
- How many things it contains (if it's a container)
- Who is using it now

In addition to these attributes, certain types of desktop objects have the ability to contain other desktop objects. Folders, for example, can contain other folders, documents, etc. Thor allows you to inspect and use the contents of any container. You can use containers to organize your desktop.

Things in the Desktop World

Disks & Disk Drives

Disks and disk drives are separate entities in the Valhalla desktop world. Disk drives are objects on the desktop which directly map to their real world counterparts connected to the computer. Disks, on the other hand, appear as separate objects on the desktop. Disks can be inserted into, or ejected from, disk drives. Disks provide storage for other types of desktop objects such as folders and documents. In addition to giving you space, disks help you to organize other objects on your desktop.

An important attribute of disks is that you need some sort of device to use and access the information on them. In order to use a floppy disk, you need a floppy disk drive. In order to use a CD ROM disk, you need a CD ROM player of the appropriate type. Of course, if you don't have a given disk drive connected to your computer then a disk of that type could never show up on the desktop.

Appearance of Disks & Disk Drives

Disks represent the various types of media you use for storing information. These include floppy disks, tapes and CD ROMs. Each real physical piece of media will be represented on your system as a disk resembling its real world counterpart. Each disk reflects the attributes and behaviors of the real media it represents. For example, locked floppies or CD ROM's are read only - they can not be modified and appear as locked on the desktop.

Disk drives appear as objects which look like the real device connected to your computer. When they have a disk inserted they take on the name of the inserted disk. Otherwise they will appear with some distinguishing name enabling the user to discern which physical device the object represents.

Generally disks will not appear separately on your desktop. When inserted into a disk drive the disk will appear as inserted and when ejected it will disappear. However, there will be a way of allowing disks to be ejected onto the desktop so that it will be possible to work with multiple disks with a single disk drive.

Use of Disks & Disk Drives

You use disks to store things. You can open a disk to reveal its contents which can include most any desktop object except other disks. You can also move the contents of the disk around within the disk and open objects on the disk in order to work with them.

You can copy one disk to another by simply dropping its icon on that of another disk or disk drive. If either disk is not inserted in a disk drive, you will be prompted to insert it in an appropriate drive. If you drop the icon on an empty disk drive, you will be asked to insert an appropriate disk. If the target disk is not large enough to hold the entire contents of the source, the copy will fail.

Specific kinds of Disk Drives

There are two basic kinds of disk drives you will see on almost all Pink systems. These are Fixed Disk Drives and Removable Disk Drives.

Removable Disk Drives

Removable Disk Drive is the name which decribes floppy disks or any type of disk with removable media. The icon for a Removable Disk Drive always shows whether or not the drive contains a disk.

Fixed Disk Drives

Fixed Disk Drive is just a fancy name for a hard disk. However, the name tends to convey the functionality quite well. Fixed Disk Drives are just like Removable Disk Drives with disks permanently inserted. The disks cannot be ejected or otherwise removed under normal circumstances. Thus, a Fixed Disk Drive always contains the same disk.

Thor Disks & Disk Drives vs. Macintosh Finder Disks & Disk Drives

Thor disks are somewhat different than Finder disks. Finder disks represent both media and drives all rolled into one. This is the source of much real confusion and a number of theoretical user interface problems. Thor makes a distinction between disks and disk drives because, after all, they are two separate things. Thor disks simply represent the storage medium itself. Disk drives represent the hardware you use to access disks.

Folders

You use folders to organize the things you store on disks. Their behavior, not surprisingly, is similar to real folders. They can contain documents, other folders, etc. They are a bit more flexible than real world folders, however, in that you can put many kinds of things into them. In addition a folder can store any number of things. It is limited only by the amount of space left on the disk it is stored on.

Appearance of Folders

Folder icons look like folders in the real world. When opened, the folder representation will be changed to reflect this.

Use of Folders

It is extremely simple to use folders to organize things. You can create a new folder on a disk or in another folder. To organize things, you simply put whatever you want into the folder.

Thor Folders vs. Macintosh Finder Folders

Thor folders serve exactly the same function as Macintosh Finder folders.

Cross-References

Cross-references in Thor are used to tie things together. By creating a cross-reference to something, you can always find it by using the cross-reference. Cross-references are very much like real world cross-reference cards. They are explicitly *not* the thing itself - they merely point you to the thing of interest.

Appearance of Cross-References

All cross-references will have a distinct appearance. Each, however, will have enough visual similarities to the thing it points to that you can recognize what it points at.

Use of Cross-References

You can make a cross-reference to any desktop object. Once you have done this, you can put the cross-reference wherever you want. Whenever you run across the cross-reference, you can use it to get at the real object. A simple example is project documents. A project document simply contains cross-references to other documents. Once you have your project document available, you can automatically get to the things in the project.

Cross-references also let you leave "breadcrumbs". By making a Cross-reference to a folder and putting it in another folder, you can indicate that the two are related and, at the same time, give yourself or someone else a quick way of getting to the related folder.

Thor Cross-References vs. Macintosh Finder Cross-References

Cross-references are very similar to NuFinder Aliases. There are, however, a few important differences. First, NuFinder aliases are one way only. You can not find the things which refer to an object. Thor defines cross-references as being bidirectional. This allows users to find out if there are any cross-references which refer to a given object. Second, Aliases exist only at the file level. Thor cross-references will work exactly like Pink hypertext links in that they can exist anywhere, including within documents.

Documents

Documents are analogous to real documents or forms. Things like spreadsheets, letters and drawings are examples of documents. Documents contain application specific information. They normally also contain some information which Thor can understand and display for you (such as cross-references). In order to create a new document you must have appropriate stationery for the type of document you wish to create. It will always be possible to get a new piece of stationery by opening up the desired application. In order to edit a document, you must have the application which knows how to work with that document.

Appearance of Documents

All documents have the same basic appearance. They have an outline which looks like a page with one corner turned down. Within this framework, each kind of document has its own custom appearance.

Use of Documents

In general, to use a document, you open it. An open document is displayed in a window. You can see and work with the information according to the rules defined by the Pink Human Interface and the specific application which lets you work with the document.

You can also use Thor to get and modify additional information about the document. When you do a Thor "Get Info", you will be shown information about the document. This will include Cross-references, or links, to other desktop objects. Thor will let you follow these connections to see what they are.

As with other objects, you are free to organize documents however you wish. You simply drag them to the desktop, or the disk or folder you want to store them in.

Specific Kinds of Documents

There are several kinds of documents which exist in the Thor environment. These include:

User Dossiers

Thor is aware that different people often use the same system. Thus, Thor will attempt to make it easy to personalize the behavior of the system by keeping all user system preferences in a single document called a User Dossier.

User Dossiers describe the people who use a Thor system. They serve several purposes. First, they allow you to indicate your personal preferences. When Thor is told that a specific User Dossier describes the current user of the system, it can adapt to that person's preferences. User Dossier's are also used for controlling access privileges and for authentication. The possibility exists that application preferences will also be kept in a User Dossier.

Address Books

Address Books help you to find and access resources external to your system. They help in keeping track of often used remote devices such as printers or disks. They can also be used to store addresses of remote systems and networks.

Projects

Project documents allow you to tie documents together in arbitrary ways without moving or altering the documents themselves. Imagine that your normal way of working involves dividing the world into a set of folders based on document type. One folder contains letters, one drawings and another scanned images. Now you need to do a project involving a few images, a drawing and a letter. You can create a Project by simply dragging a cross-reference for each document into a new Project. Now you can open them all at once by opening the Project. You never need to actually move the documents themselves. This approach allows you to include the same document in multiple projects. It also allows you to leave the original documents wherever you want them.

Viewers

Viewers let you look at the world from different perspectives. You use them to create lists of things based on what you consider important. You might be interested in everything on a disk which contains the word "dragon" and was written over a year ago. A viewer will compile a list of such things for you.

The entries in a viewer list are actually cross-references. You can use these to get at the actual desktop objects to which they refer.

The possibility of combining folders and Viewers will be explored.

Thor Documents vs. Macintosh Finder Documents

Thor documents serve the same function as Finder and NuFinder Documents. Unlike older Finders, however, Thor can display some of the contents of any document since documents can contain other desktop objects.

Although not currently specified, the Thor model of Projects could be easily implemented under NuFinder by using aliases.

Stationery

Stationery provides a convenient and natural way to create documents. You will have a piece of stationery for each kind of document available on your system. The kinds of documents available are determined by the applications on your system.

You can make custom stationery from any document. This lets you have different standard documents of the same type available. For example, you might want to make stationery for your company letterhead. This saves you the trouble of recreating your letterhead every time you write a letter.

Appearance of Stationery

Stationery looks like a pad of paper. The topmost sheet will always look like the icon of the type of document the stationery creates.

Use of Stationery

When you open a piece of stationery, you will actually create a new document. You can then edit and save the document in the normal way.

Creating custom stationery is simple. You simply edit a document until you want to make stationery out of it. Then you simply issue a "Make stationery" command. Now, whenever you create a new document using that stationery, it will be a copy of the document you used to create the stationery. You can also always get a new piece of stationery by opening up any application.

Specific Kinds of Stationery

Thor provides stationery for each of the standard kinds of documents described in the documents section.

Thor Stationery vs. Macintosh Finder Stationery

Thor stationery serves the same purpose as stationery in NuFinder. Unlike NuFinder, however, Thor stationery provides the only standard way to make new documents (other than directly copying them).

Applications

Applications allow you to edit documents. In general, you merely need to put the application somewhere in your desktop world. In some cases, you may be able to customize the behavior of some applications by opening them up and changing specific options.

Appearance of Applications

Applications will have custom designed presentations just as today's Macintosh applications have custom designed icons.

Use of Applications

You do not normally use applications directly. The presence of a given application gives your system the ability to edit specific kinds of documents.

You can open an application to find out specific things about it or to modify its behavior in ways defined by the application.

Specific kinds of Applications

Thor provides applications for each of the standard kinds of documents described in the documents section.

Thor Applications vs. Macintosh Finder Applications

Applications are similar to applications in Finder and NuFinder. The key practical difference is that opening an application in Thor does exactly that - opens it and displays its contents. Opening an application does not create a new document as it would in NuFinder or Finder. Opening an application in Thor will allow users to change application preferences as well as view and edit the components of the application.

Appliances

Appliances in Thor are analogous to appliances in the real world such as those you would find in your kitchen. In the real world appliances give you a certain way to process things. For example, a toaster "processes" toast. In Thor, they give you standard ways of performing certain tasks. For example, you use appliances to print documents or to destroy documents.

The key distinguishing feature of appliances is that they actively process documents instead of editing the contents. This processing may alter the document, it may result in hardcopy output, or it may result in a new document being produced. The exact results depend on the specific appliance.

Appearance of Appliances

Appliances look like what they represent. The Trash looks like a trash can and printers look like real printers.

Use of Appliances

You use an appliance by dropping a document on it. The appliance will then process the document. Some appliances can only actually process one thing at a time. In such cases, you can think of dropping the document onto the appliance as being like putting it into an input hopper. It will be processed when its turn comes up.

Some appliances will not accept all types of documents. If an appliance can accept a document, it will highlight as you move the document over it. If it does not highlight, you know that it will not process the document.

You can open an appliance to inspect its state and to directly control its behavior.

Specific kinds of Appliances

Thor comes with several appliances already installed.

The Trash

The Trash is the way you dispose of desktop objects you are no longer interested in. When you throw something in the Trash, it stays there until you empty the Trash.

The Trash is essentially the same as the Trash in Finder and NuFinder.

Printers

Not surprisingly, you use Printers to print documents. To print a document, you simply drop it on a Printer. The Printer will print the document and put it back where you took it from.

Thor Appliances vs. Macintosh Finder Appliances

Appliances are analogous to NuFinder Grinders.

Tools

Tools are like little utensils. They have no associated document types and they do not process documents. They are used to perform certain small, well-defined tasks. Tools may have state which is saved between uses.

Appearance of Tools

Tools look like the kind of thing they represent.

Use of Tools

To use a tool, you open it up and use it directly. Each kind of tool will present an interface which is appropriate for what it does.

Specific Kinds of Tools

Thor will provide several standard tools.

Clock

The clock represents a clock. It simply keeps and displays the time. It will also provide some support for setting alarms.

Calculator

The calculator is a simple calculator. You can do basic computations. You can cut and paste to and from the calculator.

The Macintosh

The Macintosh represents your computer and its various parts. With the Macintosh tool you can inspect and control the behavior of your system. It is represented as a small image of your computer. When you open up the Macintosh tool, you will see a diagram which indicates the parts of the system and their current operating state. You can control some parts directly such as speaker volume and the clock. Others parts of the system will need to be opened up to be altered or viewed. For example, you will see each of your NuBus card slots and the cards in them. If you wish to adjust the settings on a given card, you will need to open that card up. The Macintosh tool replaces most of the functionality of the old Control Panel and NuFinder Panels. It performs the same functions in a more concrete and direct manner.

Thor Tools vs. Macintosh Finder Tools

Tools take the place of a number of old Desk Accessories. They are meant to serve much the same function: to be quickly accessible and used for simple tasks.

Components

Components in Thor are parts of other things. Applications, for example can contain several different components such as a spell checker or fonts. The key aspect of components is that they have no functionality in and of themselves. They are only useful when they are inserted into some other kind of desktop object which can make use of them. Thor has no way to look inside of components. This is in contrast to other kinds of objects where Thor can normally show you what they contain. In general, you won't need or want to deal with Components.

Appearance of Components

Components will look like little black boxes. They will normally appear within applications when they are opened. They can, however, live anywhere in the desktop world.

Use of Components

Normally you won't directly use components. They will be visible within applications and will be moved around only when you want to upgrade or add components to an application.

Thor Components vs. Macintosh Finder Components

Components are similar to raw files (unknown document types) under Finder. They need a separate concept to represent them under Thor since Thor documents are not tied to individual two fork files as are documents under Finder.

Programming Interface Specification

Karl B. Young
Steve Horowitz
George Norman
Joel Spiegel

56

Contents

About this document. 3

Introduction and Overview. 4

Operations on Desktop Objects. 6

Presenting the Desktop Object. 10

How the Magic Happens. 11

56

About this document

This document describes Odin, the suite of classes and methods which allow all Pink applications to deal with desktop objects. Through the interface described herein, any action represented by the Finder applications (collectively called Thor) can also be accomplished from any application. In fact, Thor will be using these same classes to do its job.

This document will give the developer of a Pink application enough background to use and understand desktop objects. We do not list nor discuss actual method calls, except as examples. Nevertheless, readers should be familiar with the Pink project and C++ in general.

This is a working document. It describes what is currently known about the project. It will be updated as needed.

Document Outline

This document is split into four sections as follows:

- **Introduction and Overview**—an introduction to the Odin classes and their benefits. Basic concepts for desktop objects and the presentation classes are introduced.
- **Operations on Desktop Objects**—what you can do with a desktop object. How to create a new one. Also, a little bit about storage allocation and responsibility
- **Presenting the Desktop Object**—the way to show desktop objects in the way the user wants to see (or hear) them.
- **How the Magic Happens**—the implementation behind desktop objects, for curious folks.

Related Documents

There are three additional documents which describe various other aspects of Valhalla. These may be useful in understanding Odin and its place in the world.

- *Valhalla Project Overview*, describes the Valhalla project as a whole. Valhalla is an umbrella project which is made up of the Thor user level applications and the Odin desktop services.
- *Thor User Interface Specification*, describes the Thor user interface. This includes a description of objects on the desktop and how they are manipulated by the user. This document is of interest to anyone who wants to understand the user view of Valhalla in detail.
- *Valhalla Construction Manual*, contains the "secret wisdom" of Valhalla. This includes detailed architectural and implementation information. It is of interest to those who are involved in building or reviewing Valhalla as a whole.

Introduction and Overview

According to Norse mythology, Odin is the great magician/warrior of Valhalla, the one who orchestrates the battles of myriad brave warriors, and who conjures up the necessary magic for their success in warfare. Together with his son Thor—the greatest of all warriors—Odin sees to the continued success and welfare of the North countries.

In the Pink system, the Odin libraries and servers provide the necessary magic for all applications to successfully manipulate desktop objects. Odin orchestrates the communication between applications and desktop objects, coordinates multiple requests to the same objects, and allows for simple and consistent presentation to the user. Together with the collection of applications that make up the Finder (known as Thor), Odin has the stewardship and responsibility for the desktop metaphor and the desktop world.

Benefits of Odin

Traditionally, the Finder has been unique in the way that it deals with objects which can reside on the desktop. Where one application presents a *list* of names to choose from, the Finder presents icons to manipulate directly. Where another application has a "Save As..." option, the Finder allows the user to duplicate the selected object. Where another application has to provide a "Clear" or "Delete" option, the Finder allows the user to drag things to a trash can (and retrieve them).

The dichotomy between the Finder and "normal" applicationshas been tolerated only because normal Macintosh applications deal so seldomly with files, folders, and disks. Under Pink, however, annoyance would certainly grow into customer revolt if linking, sharing, and cooperative working were handled in an inconsistent fashion.

The tremendous effort which has gone into making the Finder easy and useful (i.e., the desktop metaphor) must be spread among all applications, if only to make the effort worthwhile. More importantly, the user receives the benefit of having a single, direct-manipulation interface to all desktop objects.

Odin provides the benefits of consistency to both the user and the developer. From the user's perspective, the direct manipulation so prevalent in the Finder is now present everywhere. Icons continue to look and act the same from place to place, and no relearning of concepts is necessary. In addition, should the user's preference for interface change (say, from graphical to telephonic), this change is reflected across the system, and not just in the Finder.

From the developer's perspective, all actions that the user performs to desktop objects have a direct corollary in Odin's interface to these same objects. All file system operations are expressed in terms of desktop objects, and their effects are immediately reflected in that arena. No longer does the developer have to think in one mode (the File System) while presenting another (the Desktop World). In addition, the standard presentation of each object is previously defined and available for use by the application. Reinventing the user interface is no longer necessary.

Overview of Desktop Objects

We use the term "desktop objects" to generally describe anything that the user can move around. The term makes sense historically as any object which the Finder manipulated can be placed on the desktop. In Pink, the definition still holds, but the emphasis has shifted from the Finder to all applications that show any manipulable objects to the user.

Desktop objects in Odin reflect the attributes and contents of the real "thing", such as the trash, a document or a disk. They model the behavior of the desktop object. All information about a desktop object obtained through the Odin interface will map to some concrete attribute of the real object. These at-

tributes run the spectrum from very concrete (geographic location, say, of a printer) to much less concrete (color of a document when presented on a monitor). Any changes to these attributes made through Odin will make a real change to the desktop object that is represented.

Each desktop object can be thought of as a potential container of other desktop objects. This is familiar to users of the current Finder, especially in the case of folders and disks. This concept now extends to all desktop objects. Even documents may contain, say, a cross-reference to another document, or a specialized spelling database that can be dragged to other documents.

The corollary to "containership" is that every desktop object is contained by another desktop object. We get around an infinite hierarchy by having one exception: the Desktop itself. We define the Desktop to be a "desktop object" but with the key exception that it does not have a container (or home) itself. This, of course, implies that the Desktop itself cannot be contained by other desktop objects.

Presentation of Desktop Objects

The programmatic interfaces provide the application developer with classes which model the behavior of desktop objects. This is only one side of the coin that applications must deal with. The presentation of the desktop object to the user is at least as important as their attributes. Odin therefore provides a set of classes for selecting, presenting, and manipulation of desktop objects.

The idea behind the presentation classes is to facilitate presentation of desktop objects for Pink applications as well as to encapsulate knowledge about desktop object display in one central location. Selection of the correct presentation will take advantage of knowledge about the user to make intelligent decisions about what kind of presentation to use. Thus when the user changes his preferences all applications which make use of desktop objects will change their presentations accordingly. Thus Odin will provide the facilities for consistent representation of desktop objects across all Pink applications.

Operations on Desktop Objects

When dealing with desktop objects through Odin, there are some basic concepts and rules which must be understood. These "desktop axioms" provide a framework for operations on desktop objects. They deal with how desktop objects deal with those who make requests of desktop objects, concurrency, and the side effects of operations.

The desktop axioms are listed below.

- Any task can make a request of a desktop object. The corollary to this is that multiple tasks may be making concurrent requests of a single desktop object.
- The requesting task blocks until its request completes. If a particular application wishes to make concurrent requests to objects, then it is necessary to spawn multiple tasks.
- At completion, a request has either succeeded or failed. If it succeeded, the request was carried out completely and may have resulted in side effects. If it failed, the request was not carried out and resulted in no side effects.
- A valid key is delivered as a result of a successful acquire.
- A side effecting request, other than acquire, must include a valid acquire key or it will fail. Acquire privileges may be shared with one or more clients by cloning or sharing the key.
- An acquired object must be released before another acquire to that object can succeed.
- If an object is unstable (that is, being changed), all additional requests may *queue* until the object becomes stable. A FIPO queue is used (First-In-Priority-Out, where the exact priority scheme is still being decided) to queue all conflicting requests.
- If an object is engaged (that is, being read but not being changed), all additional side affecting requests may *queue* until it becomes disengaged. Additional non-side affecting operations will not block if there are no side affecting operations pending in the queue. Additional non-side affecting operations will block if there are side affecting operations pending in the queue.

Operations Available For All Desktop Objects

There are many operations which desktop objects may perform. They are described in detail in the header files for the Odin classes themselves. Below we list some of the general operations available to all desktop objects, i.e., the methods supported by the most abstract desktop object.

- **Acquire/Release**—Before any change can be made to a desktop object, it must be acquired. Acquire will return an "acquire key" by which side-affecting operations may be authorized. It is not possible for anyone else to acquire an object until the particular key is released.
- **Lock/Unlock**—Make a desktop object write protected or not.
- **Move**—Move a desktop object to a new position within its container.
- **Dispose**—Dispose of a desktop object completely (note: this is not the same as moving a desktop object to the Trash object).
- **GetContents**—Get the contents of this desktop object.
- **GetContainer**—Return the container in which I reside. This will succeed for all desktop objects, except the Desktop itself.
- **GetName/SetName**—Get or set the name of the desktop object.
- **GetComments/SetComments**—Get or set the comments associated with a desktop object.

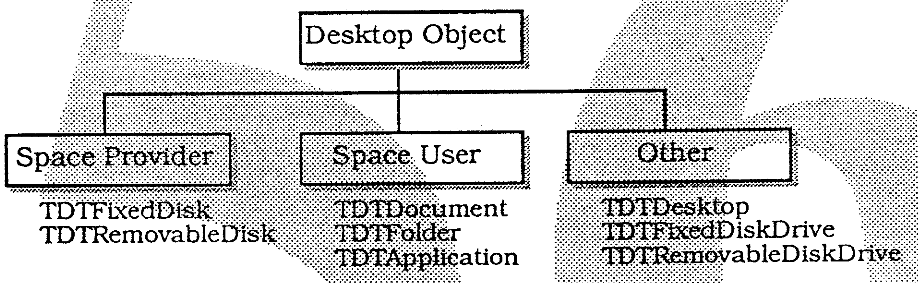
- **AddDependent/RemoveDependent**—An application may add a dependent task to a desktop object. Whenever an attribute of the desktop object changes, the task is then notified of the change. Dependents may be added based upon particular attributes.
- **GetAttributes/SetAttributes**—Get or set the specific attributes associated with a desktop object.

Besides the operations available to abstract desktop objects, specific operations may be added by classes derived from desktop objects. As examples, a document may be moved from one container to another (within the same disk), a disk may be unmounted, a disk drive may eject a removable disk.

Sample Odin Classes

Odin defines classes which represent real desktop objects and make them accessible from all applications. The desktop object instance which we manipulate within an application is a representation (or "surrogate") of the real desktop object (say, the disk or document). This implies that any change to a desktop object by Application A must be immediately visible in Application B.

Desktop object classes within the application fall conveniently into three groups: 1) objects which provide space (say, a disk), 2) objects which use up space (say, a document or folder), and 3) everything else (for example, a disk drive). This is partially illustrated below.



Although this is not an exact hierarchy of classes (which is being updated as you read this), it will serve for our examples (the abstract classes will be called TDesktopObject, TSpaceProvider, TSpaceUser, and TOther for now).

Using Desktop Objects

Most desktop objects will be provided for you by other objects. The one exception to this is the desktop itself. It is declared simply as follows:

```
TDesktop theDesktop;
```

The desktop can then be used to find out about other objects (disks, etc.) that may be contained in the desktop. In the future, it may be that the application framework will provide other default desktop objects. For instance, an application which opens a document may be provided with the desktop object that represents that document.

Once you have an instance of a desktop object, you can query it for information about itself, its container, and the objects it contains. For instance, we can query the desktop what objects it contains.

```
TDesktopObjectSet theContents;
theDesktop.GetContents(theContents);
```

The set of desktop objects returned by GetContents is a snapshot of the contents at the time the call was made. Since other applications (or even other tasks within the same application) could be updating the

contents as soon as the GetContents call is done, there is no guarantee that theContents is eternally accurate.

Assuming a relatively stable world, the following would allow a recursive descent through every desktop object in a system.

```
void descend(TDesktopObject& thisObject)
{
    // This is the point at which you would do something to the object
    // say, print its name, whatever
    DoSomethingTo(thisObject);

    // get ready to descend
    TDesktopObjectSet theContents;
    thisObject.GetContents(theContents);

    // iterate through the contents
    TDesktopObjectSetIterator theIterator(theContents);
    TDesktopObject aDesktopObject;      // stick the desktop object here
    aDesktopObject = theIterator.First();  // get the first one

    while (IsValid(aDesktopObject))
    {
        descend(aDesktopObject);                    // go down hierarchy
        aDesktopObject = theIterator.Next();        // get the next one
    }
}

TDesktop theDesktop;
descend(theDesktop);
```

Creating a New Desktop Object

So far we have discussed accessing existing objects in the desktop object world. Creating an entirely new desktop object from an application is possible but has some additional constraints.

In order to create a new desktop object, it is necessary to provide enough information so that the new object may safely live in its new home. Therefore, it is at least necessary to specify the desktop object in which the new object will live. Most objects also expect a name or other specification so that it can be created unambiguously.

The actual creation takes place within the application by specifying all the necessary information to a constructor of the type of object you wish to create.

```
// create a new folder in parentObject with the name "New Folder"
TDTFolder newFolder (parentObject, TText("New Folder"));

// create a new text document on the desktop with the name "Read Me"
TDTDocument newDocument(theDesktop, TText("Read Me"), DocType(kTextDoc));

// connect a new hard disk to the desktop by specifying SCSI port
TDTFixedDiskDrive newDisk(theDesktop, IOSCSIPort(kSCSI5));
```

It is not possible to create an "abstract" desktop object directly. There is no constructor, say, for creating a generic Space User which would be specified more concretely later.

Desktop Objects in the Application

In order for a desktop object instance to be useful, they must be lightweight and easily replicated. Once you are given an instance of a desktop object class, you can access it, duplicate it, and transform it into more or less abstract classes. All these instances will still refer to the same "real" desktop object. Since these classes are very lightweight, there is little penalty for the duplicate versions.

```
// we are given "anObject" of type TDesktopObject

TDTFolder aFolder = anObject;    // will fail if anObject is not a folder
TDTFolder aFolderToo(anObject); // a faster way to do the same thing

TDesktopObject object2;          // an "empty object"
aFolder.GetContainer(object2);   // GetContainer fills in object2

TDTDisk aDisk(object2);          // make it a disk(exception if not a disk)
aDisk.Erase();                   // and erase it
```

In the above example, there are three instances referring to the folder object, and two instances referring to the disk object. This is possible because each instance is "referring" to the real object, and any change made by one would be reflected in others that refer to the same object.

Space Allocation and Responsibility

The objects in the Odin classes are designed to be as simple as possible regarding memory allocation. We recommend that instances be allocated on the stack whenever possible. It is always better to pass an instance of a desktop object by value or reference rather than by a pointer. All the appropriate methods are supported for converting between different kinds of desktop objects (for example: TDTFolder to TDTSpaceUser to TDesktopObject and back).

The general rule is that you own the storage for any instance of desktop object that your get your hands on. This is evident in the TDesktopObjectSet and TDesktopObjectSetIterator classes. Unlike the more general collection classes, they return copies of the items collected within them. In like fashion, the TDesktopObjectSet will dispose of any storage associated with its contents when the set itself is deleted. Any instance of desktop object garnered from the set before its destruction is still valid, however.

Presenting the Desktop Object

The presentation classes will ultimately exist as a shared library accessible to any pink application. Applications will make use of the presentation server as a means to decide what type of presentation class to instantiate. An application will give the presentation server certain user information, attributes and characteristics which the presentation server will use to make a decision about what type of presentation class to pass back.

The application will be able to specify arbitrary attributes for use in the process of selecting a presentation. The presentation server will then pass back some sort of identity which will specify a given presentation from the global presentation database. The application can then instantiate a presentation of this type. If the presentation server passes back more than one type of presentation identity then the application can pass back each of the types which it was given along with some more information to allow the presentation server to narrow the choices down to a single one.

Following is an example of this process in action. Suppose a given window is displaying a series of tools by icon. If the user then selects "View by Name" the application will then make a call to the presentation server for each of the objects in the window. This call to the presentation server will specify all of the attributes which the application wishes to present. In this case it would be name only. The presentation server would then pass back the identity of a presentation which specifies a text-only representation of the object.

The application would then instantiate an object of the new presentation type and the user would see each of the objects presented with name only. Other attributes which could be passed into the presentation server would be icon(graphic), lock state, busy status, etc. The presentation server will use these attributes along with information about the user to make an intelligent choice about what type of presentation class to pass back to the application.

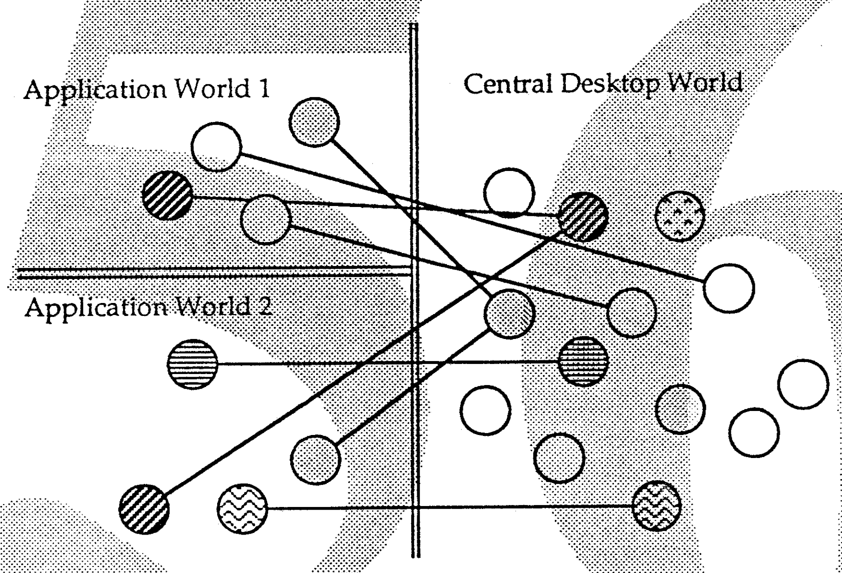
To facilitate simple use of desktop object presentations the presentation server will have some set of "standard" graphic and text presentations which can be requested with some minimum amount of information by an application.

How the Magic Happens

Throughout this document we have emphasized that the Odin classes create instances which refer to the desktop object, but are not the object itself. This section goes into more detail as to how this magic is carried out. NOTE: It is not necessary to know or understand this section in order to use the Odin classes.

There exists on each system a "desktop server" whose job it is to manage the requests made to desktop objects. Every request that is made by an application to an instance of an Odin object gets forwarded to the desktop server.

For each item that can appear on the desktop there is a specific central desktop object within the desktop server that provides the information about that item. These desktop objects provide the real link between the "real thing"—a disk, a file, a printer—and the information returned to applications. Central desktop objects are not generally concerned with how information about the object is presented, but rather with the accurately reflecting the state of whatever it is they represent. Central desktop objects provide the model of the state and behavior of things that represented on the desktop.



Requests made to desktop objects in the application world get forwarded to the "real" object in the central desktop world.

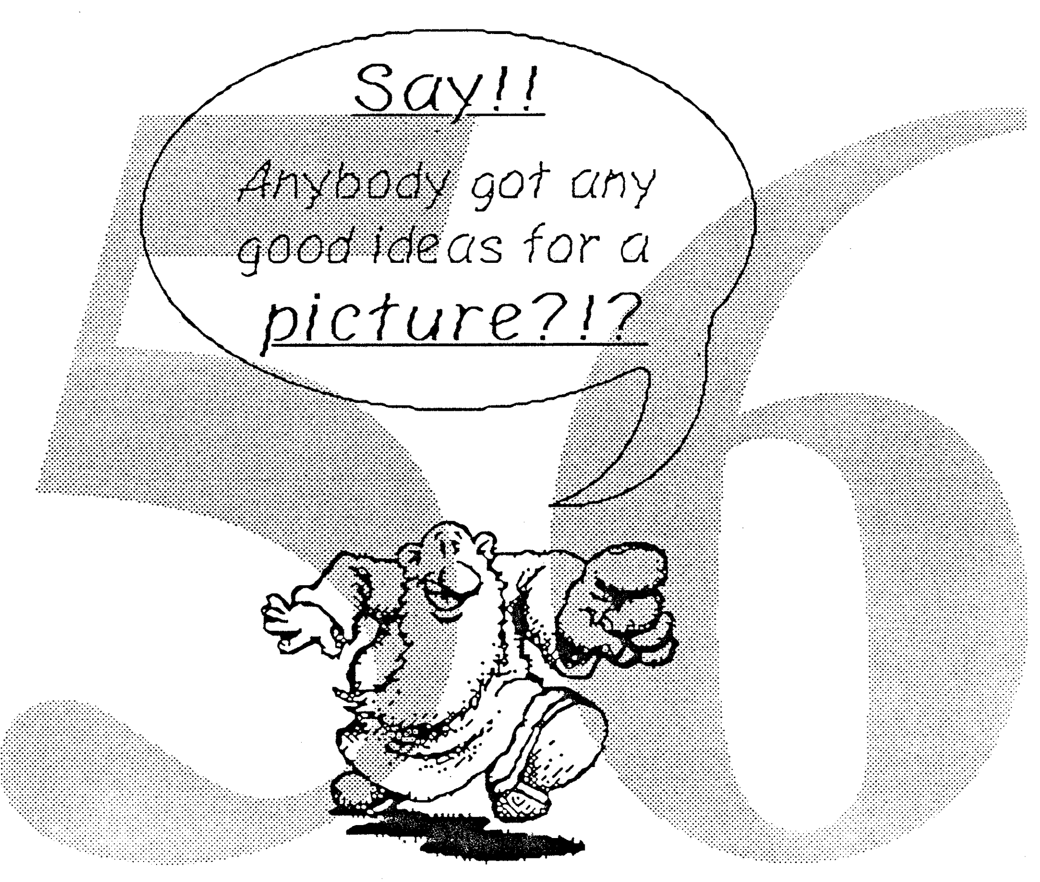
While each request is made synchronously at its source, the desktop server must be able to handle multiple requests asynchronously in order to serve multiple tasks and teams. Therefore, each request that comes to the desktop server gets handed off to a separate task, which then makes the request of the appropriate central desktop object. When the results of the request are known, the task replies back to the application with the results.

56

Mood Indigo
The Blue Adapter

56

Mood Indigo
The Blue Adapter



by
Rick Daley, Phil Goldman, and Rick Holzgrafe

56

Introduction

The Blue Adapter is a crucial piece in the Pink system. It will allow Pink users to run most of the large base of Blue applications, as long as the target machine is a Macintosh with a 68xxx-family CPU (as in every Macintosh shipped to date). It will also support many INITs, FKEYs, DAs, and CDEVs. The adapter may not support Blue code that must run at interrupt time in order to function correctly. It therefore may not run some drivers nor applications that depend on such code.

The adapter is not a window onto an underlying Blue system and/or ROM. Rather, it is an emulation of the Blue application execution environment. Therefore, features added to Blue must be explicitly added to the adapter as well; they will not magically appear.

With the addition of MMU support, via Opus/2, the adapter will provide for more protection and a potentially larger address space. Efficiency will be virtually indistinguishable from that of a native Blue system.

Architecture

After evaluating several alternatives, we have settled on an "Extended MultiFinder" model (similar to the approach taken by A/UX 2.0). We take most of the existing MultiFinder code and simply run it as a Pink application, in a single Opus/2 address space. MultiFinder itself is implemented as a team in which the major task runs the bulk of MultiFinder code while a small set of other tasks handle VBL interrupts, "low-memory" maintenance, and so on.

The adapter does its own scheduling of Blue applications, rather than deferring to Opus' pre-emptive scheduling. Blue-style co-operative scheduling is accomplished by making each Blue application a light-weight task sharing memory with MultiFinder and the other Blue apps. (Sound familiar?) Blue app tasks block on an Opus IPC message from MultiFinder. They return the message to MultiFinder when they are willing to "surrender the CPU". MultiFinder treats this "unblock-and-run" message as a token which it passes to each Blue app task in turn.

The leveraging of MultiFinder code provides a strong foundation. Applications will find themselves in an environment that emulates as closely as possible the one they were designed for. DAs, INITs, and drivers (those which don't try to directly access hardware) should run with little or no special support from the Blue Adapter. If the Adapter emulates Blue capabilities at a relatively low level, new features from System 7.0 and later Blue releases should be easy to add.

We currently assume that the important Blue applications will be 32-bit clean by the time the Blue Adapter ships. If not, there are options we can pursue. (It might be possible to run 24-bit applications along with 32-bit ones, by running all the 24-bit applications in the first 16M, never showing them addresses ≥ 16M, and handling the faults caused when they put garbage in the high byte of dereferenced pointers by giving them the access from the stripped address.)

There is a limited amount of protection that can be provided in one address space. Applications can have their particular heaps protected, but the rest of the space (including the Blue "low memory" image, the system heap, and temporary memory) cannot. There would be a loss of efficiency, since the access levels on the heaps must be changed at every switch. The adapter itself will be more complex and thus more difficult to extend. A better solution might be to have 256 shared areas that represent the same 24-bit world.

Implementation Issues

Extensions

The following are services that the Blue Adapter may provide, but which are not critical to the adapter's success:

- Run INITs when the adapter is launched.
- Support accessto memory-mapped hardware, either by actually mapping in the I/O memory space, or by trapping exceptions and simulating the access in software.
- Support all the private MultiFinder calls, to support SADE and shady third-party stuff (e.g. Suitcase).
- Create a wrapper to allow some Blue drivers (e.g. video and SCSI disk drivers) to work with the Opus I/O model.

Evolution Strategy

One of the great difficulties in implementing the Blue Adapter lies in the fact that more than any other part of the Pink project it is critically dependent on other parts of system software, Pink and Blue alike. Therefore the strategy for implementation is dependent on these external factors. Worse yet, some of these critical pieces may never arrive. (For example, if the body of existing applications is *not* 32-bit clean when Pink ships, we may have to emulate the 24-bit environment.) So, until we see the future, we must make some assumptions about it. The following strategy will currently suffice:

- **Promote the standard for 32-bit clean applications.** This has been in progress for some time, to prepare the world for the advent of System 7.0. The excitement in the user and developer communities about System 7.0 promises that the major Blue applications will be 32-bit clean well before Pink ships.

- **Evolve the adapter as Pink system services become available.** These services include the Event and Layer Servers, the new I/O model, new drivers, and the Pink file system.

- **Track and advise the development of new Blue software.** As the major goal of the Blue adapter is to run the bulk of Blue application software, it must provide the bulk of the Blue interface. During the entire development of the adapter this will be a moving target. It will continue to change even after the adapter release. As the Blue model continues to evolve it becomes more likely that it will push the interface past the point at which it can be supported by the adapter. We must not only follow these developments, but also advise the developers. The System 7.0 IPC interface is an example of such evolution, where advice from the Pink team is helping to shape Blue development for future Pink compatibility.

External Dependencies

The Blue Adapter is a high-level Pink application. Like any such application, it has dependencies on the services provided by the Pink system.

The adapter needs the following kernel level services:

- A timely method for receiving A-line exceptions.
- Emulation of privileged instructions.
- Reception of bus errors, in some form, for exception handling.
- A method to map in ROM and screen memory.

- A timer service, to support the Retrace Manager and the Time Manager.
- A way to synchronize with the hardware, most obviously the vertical retrace of the monitors.

The following services provided by the Pink Toolbox (shared libraries or servers) are necessary so that the adapters can share common resources at this level.

- A method to get cursor coordinates, button state, and keyboard map.
- A method to reserve space on the virtual screen for windows.
- A method to lock the entire virtual screen for out-of-window drawing.
- An HFS server.
- A method to construct the low memory that applications expect to see: VCB queue, FCB table, WDCB table, etc.

56

Scorpion

N&C Blue Adapter

56

Coping with Dr. StrangeBlue

or

How I learned to stop worrying and love Pink.

56

Jim Mathis & Dean Wong

56

Why Read This?

This document is an introduction to the capabilities and features of the Blue Adapter N&C extensions and their ramifications for non-Pink N&C products. It is intended that this document provoke discussions about the services that must be provided by the Blue Adapter to support key, existing N&C products that will not be rewritten for the first Pink release and hopefully to head-off any design/implementation decisions that would doom a key product to not work under the Blue Adapter, and thus, under Pink. As new N&C products are planned and designed, it is important that consideration be given as to how the product functionality can be provided under Pink.

What is Dr. StrangeBlue?

Dr. StrangeBlue is the code name given to the Blue Adapter effort of Pink (see preceding document). The actual implementation effort is divided into two teams: one group from system software handling the emulation of the Blue 7.0 environment and a second group from N&C handling the emulation of the Blue networking and communication functions.

Briefly, the Blue Adapter will run most, but not all, Blue applications without any changes on a 680x0 Pink CPU. It may also run some INITs, FKEYs, DAs, and CDEVs, the exact details of which are to be defined. The Blue Adapter thus provides substantial binary compatibility between Pink and Blue for 680x0 machines. The Blue Adapter does not provide software emulation of the 68000 instruction set and thus will not be available on any non-680x0 machines (e.g., RISC machine). Support for 680x0 co-processors on RISC platforms, while possible, is not within the current scope of the Blue Adapter effort.

Why do I care?

The Blue Adapter is critical to the Blue-to-Pink migration strategy for N&C. During 1989, 1990 and 1991, N&C will be delivering a wide range of connectivity products targeted for Blue. Without the Blue Adapter, we would be faced with a need to either reimplement almost all of this software or to suffer with an inadequate set of Pink N&C products for several years. Since we have just achieved a critical mass of N&C products for Blue, it is unthinkable to backslide when Pink is released.

The Blue Adapter allows us to rewrite only a portion of our product base for Pink and plan for the gradual migration of all N&C products to native Pink. In theory, all of the MCP-based products, Comm Toolbox tools, and actual network applications should run unmodified on the Blue Adapter; that leaves reimplementation of only driver-level code, which is a substantial effort in its own right.

In practice, the transition won't be quite that easy. The Blue Adapter will support the defined *Inside Macintosh* interfaces. Unfortunately, to get many N&C products to function at all or with acceptable performance, it is necessary to use system-level knowledge that is not standardized in *Inside Macintosh* and that heavily depends on the actual underlying Blue implementation. One of the biggest challenges of the Blue Adapter effort will be to find those hidden assumptions and to accurately emulate the implementation of Blue.

Details...Details...

The exact details of the Blue functions (documented otherwise) faithfully emulated in the Blue Adapter will evolve as the Blue Adapter code is completed. At this early stage, it is impossible to give a definitive definition for what will and won't work under the Blue Adapter. This section provides the current knowledge about how to code a product to be binary compatible with the Blue Adapter.

What absolutely won't work!

Pink operates in 32-bit addressing mode.[1] All code that works under the Blue Adapter must be 32-bit clean. Luckily, most products will be tested for 32-bit cleanliness as a part of moving to System 7.0.

Code running under the Blue Adapter will NOT have direct access to any hardware devices (such as slot space) and cannot directly field interrupts. All code that directly accesses hardware must be Pink native.

What may work???

The level of support for INITs, CDEVs and DAs is currently undefined. It is not clear whether or not CDEVs such as QuickMail will work in the Blue Adapter.

System 7.0 Finder extensions will probably not work under the Blue Adapter since there is only one Finder, the Pink Finder, running on a Pink system.

What should work?

The Blue Adapter will emulate the Blue device driver calls that provide access to AppleTalk services, in particular, the .MPP, .ATP, .XPP and ADSP drivers. The Blue Adapter will also emulate the .AIN, .AOUT, .BIN, .BOUT serial driver calls. Support for other N&C device drivers is not currently planned or staffed.

The Communication Toolbox will be usable with the Blue Adapter. The amount of rewrite to the Tool Managers is yet to be determined. Tools that adhere to the published interfaces should work unchanged.

The A/Rose IPC manager will be usable from within the Blue Adapter. Software that directly accesses NuBus card memory without going though A/Rose will not work.

The VBL manager will be emulated for applications that require timing.

---

[1]Early Pink systems attempted to run in both 24-bit and 32-bit mode. The time it took the PMMU to switch between modes at interrupt level caused the LocalTalk SCC to overrun with resulting loss of AppleTalk packets.

Functional Overview

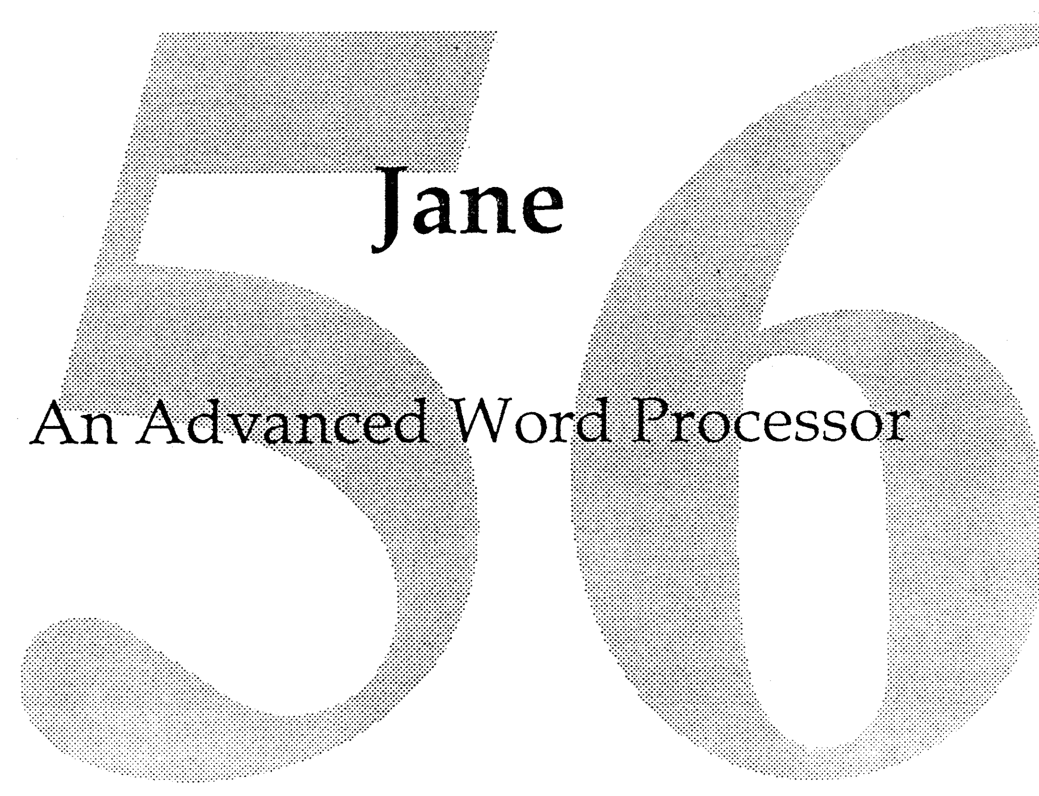
Function/Interface Call	Calls Supported	Notes and Limitations
.MPP Driver	writeDDP, closeSkt, openSkt, loadNBP, confirmName, lookupName, removeName, registerName, killNBP, setSelfSend, unloadNBP	Full AppleTalk support except for direct access to LocalTalk LAP (i.e., writeLAP, attachPH, detachPH).
.ATP Driver	relRspCB, closeATPSkt, addResponse, sendResponse, getRequest, openATPSkt, sendRequest, relTCB, killSendReq, killGetReq	None.
.XPP Driver	openSession, closeSession, abortOS, getParms, closeAll, userWrite, userCommand, getStatus	Only ASP calls are supported. Direct AFP calls will not be exposed to Blue Adapter applications.
ADSP Driver	TBD	Will support extra calls for secure ADSP streams.
LAP Manager	L802Attach, L802Detach	Only calls to access 802.2 protocol dispatcher are supported.
.ENET Driver	EWrite, EAttachPH, EDetachPH, ERead, ERdCancel, EGetInfo, EAddMulti, EDelMulti, ESetGeneral	Depends on the Blue Adapter to emulate Slot Manager calls.
TokenRing	N/A	All TokenRing services are accessed by direct messages sent via A/Rose IPC.
A/Rose	TBD	
.ipp Driver (MacTCP)	None	Requires Pink TCP/IP module.

Function/Interface Call	Calls Supported	Notes and Limitations
Serial Driver	TBD	Depends on Pink support for SCC handling & mode control.
Comm Toolbox Manager	TBD	Scope of CTB manager changes unknown at this time.
Serial Tool	All	Comm Toolbox connection tools should work under the Blue Adapter without change.
ADSP Tool	All	No tools changes; integration testing required.
MacTCP Tool	None	Requires Pink TCP/IP module.
LAT Tool	TBD	

Open Issues

1) The Blue Adapter file system emulator may need direct access to the Blue AppleShare client code for efficiency. Otherwise, the HFS calls would be mapped into Pink FS calls which would then be mapped back into HFS-AFP calls when accessing a Blue AppleShare server.

2) Emulation of NuFinder extensions might allow AppleMail client software to run on Pink with minimal modifications; is this feasible.



Jane

An Advanced Word Processor

56

Jane

An Advanced Word Processor
by Margie Kaptanoglu, Ann Greyson, Jeff Hokit

Overview

Jane is an easy-to-use advanced word processor designed to address the writing needs of professional writers and academicians. Two examples in the former category of writer include novelists and technical writers; some examples of the latter are a student writing his/her dissertation or a professor producing a work of research. The focus of Jane is to allow the power of the computer to facilitate the writing process, or in other words, the process of going from an idea to a finished document. Jane approaches the problem of supporting the writing process by making the assumption that producing a document generally consists of some combination of the following four actions, not all of which are performed by the author:

- **Individual authoring.** The writing process (as defined by English 101) generally consists of the following steps:
  1) note-taking, or information gathering
  2) outlining
  3) first draft (free-form flow of ideas)
  4) second draft (better organization and some polishing)
  5) third draft (final version)
- **Co-authoring.** Commonly there is more than one author involved in producing a document; for example, a group of professors collaborating on a single work of research.
- **Editing.** Most writers expect that their documents will be edited by another person or persons at some time, by an editor or by a co-author, for example. The editing could range from simple stylistic changes to rewriting an entire chapter.
- **Soliciting and incorporating feedback.** Many authors solicit opinions from their peers by asking them to read their document and make comments. This kind of opinion-gathering could occur at any time during the writing process. Most authors use this feedback to further refine their documents.

Observation has shown that word processor users are still very bound to paper. Editing almost always takes place on a paper copy of the document. Authors soliciting feedback generally distribute paper copies of their document and ask the reader to scribble in the margins. Even individual authoring is not always entirely electronic; many writers still type in a draft of their document, print it out and mark it up on paper, and then enter the changes on the computer. Ideally, Jane will make all aspects of producing a document so simple to do on a computer that the use of paper will be limited to printing out the final version of the document. There are many ways in which the tasks writers have been performing on paper can be made much easier by a computer, and we list some examples of how Jane will do this below:

- Editing capabilities such as delete, insert, cut and paste have always made editing simpler on a computer than on paper. Through the use of direct manipulation, Jane makes editing even easier by allowing users to simply grab a selection of text and move it anywhere in the document.
- Paper users have always been able to look at non-consecutive pages of a document at the same time (and as many pages as they like). This is an area in which word processors have traditionally fallen short. Jane provides this functionality by allowing the user to open as many views of a document (or of many documents) as desired. Jane also lets the user zoom in to, or out from, a document. Zooming in allows the user to read or edit some text at a larg-

er point size than it is written in; this is useful since text formatted in a small point size is often difficult to read on the monitor. Zooming out lets the reader view a larger portion of the document at one time.

- Tasks which are extremely tedious on paper, such as indeces, glossaries, bibliographies, footnotes, endnotes, headers, footers, tables of contents, page-numbering, and references, are generated and maintained automatically within Jane. We are aware that many of these features are currently offered by other word processors, but in Jane we will make their usage more consistent and easy to learn. We also plan to automate such features as indeces and tables of contents to a greater degree than existing word processors have done.
- Co-authoring requires that it be possible to break a document into sections which each author can work on separately. Most word processors require that a document be a single file in order to be able to generate page numbers, table of contents, and so on. Jane allows the user to break up a document into separate units (which could be considered chapters or sections, for example), while still maintaining knowledge of the document as a whole. This makes automatic page-numbering and other document-wide features possible across sections.
- Editing a document on-line has all the advantages of writing the document on-line, since the editor can make changes in his/her copy of the document, rather than simply indicating changes. In addition, Jane enables the author to automatically apply modifications made by the editor to the original document.
- Some advantages can be gained over paper by allowing reviewers of the document to enter notes within the document itself, by making a selection and then attaching a note to it. The reviewer's job is easier because of the editing capabilities which allow him/her to modify or delete his/her note, as opposed to scribbling over it on a piece of paper. The author's job is easier because s/he can decide how to view the notes (for example, in the margin or in separate views) or hide them altogether. Also, authors can view notes selectively, by author, date or subject; something which is cumbersome if not impossible to do on paper.

We would like to emphasize that the focus of Jane is on writing the text, not designing its appearance on a page. Tools are provided which make it easy to lay out a page attractively using Jane, but our focus is not to provide highly sophisticated page-layout capabilities. Given our desire to make Jane a cohesive and well-focused application, and our constraints on time and resources, we intend to concentrate on creating a full-featured word processor, as opposed to a combination word processor and page-layout program.

Motivation

This is the section which answers the question "Why a word processor for Pink?" Our answers are:

- Writing is one of the most basic functions performed on a computer, and therefore any new system needs a word processor.
- Word processing is a major area in which we can show what is better about Pink. Annotation and collaboration are two areas which have barely been touched by existing word processors. In supplying these features, Jane will show off some of the best of Pink and set an example for future Pink applications to follow.
- The successful development of a "real" (i.e., shippable) application on the Pink platform will verify that Pink is indeed a viable system.

User Interface

As stated in the overview, one of the major goals of Jane is to simplify the writing process by making the word processor easier to use. To accomplish this goal, some improvements need to be made to the user interface model. An example of this is our use of direct manipulation text, described below. Moreover, entirely new interface designs are necessary for features which have not been offered before, such as the book metaphor, collaboration, and all types of linking. Designs for some of these new features, which will affect all applications, not just Jane, are currently being investigated by the Pink Human Interface team, as well as by the Jane team. See future versions of this ERS and the Human Interface Architecture for descriptions of how Pink plans to solve these problems.

Guidelines

We have come up with a list of guidelines for Jane to help enhance the ease-of-use and maintain consistency across Pink applications. We recommend these as guidelines to be used system-wide.

- Use the standard Pink interface for the feature, if there is one.
- Use direct manipulation actions wherever it contributes to the ease of use.
- Minimalism is our byword. Keep tools, commands, concepts, etc. to a minimum.
- Avoid the use of dialogs. Let the user modify the page directly, whenever possible.
- Display all actions real-time whenever the speed is not so slow as to be prohibitive.
- Use color in a meaningful way.
- What you see on the screen is what you get when you print.
- Where it doesn't cconflict with any of the above, make the design as beautiful as possible.

Direct Manipulation Text

One important way in which we hope to make Jane easier to use than other word processors is to provide direct manipulation text. This is an extension of the metaphor which has always been used for graphic objects on the Macintosh; i.e., objects can be selected and dragged to a new location while the mouse is held down. The metaphor is extended by making it possible to "pick up" and drag a selection of text.

Direct manipulation text works in the following way. When the user makes a selection of text, a small handle appears on the left side of the selection. The user may then click on this handle, and while holding down the mouse, drag the selection to a new location within the text. When the mouse is released, the dragged text is inserted at the new location. While the text is being dragged, a small caret is shown underneath the selection, indicating where the selection will be inserted if it is dropped. When the selected text is first "picked up", a blank area is left where it was. No reformatting of the text occurs until the selection is released. The text may be dropped in the blank area, in which case it is simply restored to its original location. When the text selection is longer than twenty characters, the selection is shown truncated when it is picked up, with an ellipses at the end. The selection is drawn opaque, with highlighting, and with a see-through shadow. This gives the text the appearance of having been picked up; it looks as though it's hovering slightly above the page. A sample is shown on the next page. The word "matter" is the selection being dragged.

It's not a question of where he grips it,
it's a simple matter of weight-ratios ...
a five-ounce b[illegible] not hold a
one-pound coconut. Look! To maintain a

Program Description

What follows is a partial description of the basic features of Jane. The motivation for our design choices has been to select the methods which will best support the writing process as described in our overview, while still ensuring that the program is simple to use. This section is by no means complete. We are currently involved in coming up with a complete design for Jane, which will be outlined in our next version of the ERS.

The Text

Jane supports all the wizzy text features that are being supplied by ZZText[1]. This means that users can create beautiful text. An issue of concern to Jane, however, is that users not be overwhelmed by all the power given them by ZZText. Our focus is to keep the interface as simple as possible. More on this in the near future.

Pages and Views

Jane follows the "toilet paper" model used by most word processors; i.e., pages scroll vertically in the document in consecutive order. A page looks like a physically separate piece of paper as in FullWrite. Pages may also be viewed in the side-by-side manner used by MacWrite II. In this view, each pair of pages is viewed side-by-side.

Multiple views of the same document are supported. For example, the user can bring up a view of pages two and three and of pages eight and nine at the same time. One view could be toilet paper style, the other could be side-by-side pages; one could be zoomed in, the other could be at 100%. Editing changes made on pages two or three which affect pages eight or nine cause the view of pages eight or nine to be updated as well. Updates are as smart as possible, to avoid too much screen redraw.

Zooming in and out is supported. This includes manual zooming (zoom to 50%, 200%, etc.) as well as automatic zooming (make this page fit within this view, for example).

Like typical Macintosh word processors, when a document is first created the user is presented with the representation of a blank sheet of paper, page one. The user may immediately begin typing. As the user types and fills a page, subsequent pages are generated automatically.

1. See Text, section 2.7.

Rulers

Jane provides "real world" rulers, as in MacDraw II, along the top and left sides of a document. These rulers may be shown or hidden, as the user desires. When the mouse is held down anywhere within the document, its position is shown within the rulers by some small indicator (like the gray lines drawn in MacDraw II). These rulers will aid the user in laying out pictures, lining up columns, positioning tabs, and so on. These rulers do NOT contain information in them about tab and margin settings, or anything else, as MacWrite rulers do.

Jane also provides something similar to MacWrite rulers, which we will call formatting bars. These contain the tab settings, the left and right margins, paragraph indentation settings, and column settings. Formatting bars can be inserted on a per-line basis. All of its settings can be changed using a direct manipulation action (grabbing the margin indicator and dragging it, throwing a tab away, and so on). There will be an automatic way of specifying the number of columns, but there will also be column indicators within the formatting bar. A column indicator can be dragged by the user to make a column larger or smaller. In this way, columns of differing widths can be created using Jane.

Sidebars

Jane makes it easy for the user to insert pictures or text within the document. Tools are provided for drawing shapes, which can be used as graphic objects, can have a picture pasted into them, or can have text typed into them. So far, we plan to provide the following shapes: rectangle, oval, and polygon. After the user selects one of these tools, s/he can click anywhere in the document and draw the shape. By default, the surrounding text makes way for the shape (wraps around it), although the user may turn wrapping off, and make the shape see-through, if so desired. Shape attributes, such as the color and size of the frame and the fill, can be set by the user.

Shapes can be selected at any time the selection tool is active, and moved or resized. Shapes may be drawn on top of one another, and may be opaque or see-through. The order of the shapes may be modified with Move To Front and Move To Back commands.

Text may be entered into a shape by simply clicking within a shape and typing. Data may be pasted within the shape by clicking within the shape and then doing a paste operation. Text or graphics (or other types of data) may be pasted in.

Shapes come with small handles at their bottom right corners. The user can grab the handle and drag it to another shape. This binds the two shapes in the sense that any text overflowing from the first shape will flow into the connected shape. A line from the bottom right corner of one shape to the top left corner of another is used to indicate which shapes are connected to which other shapes.

Searching

Search and replace functions are provided by Jane. These functions can be used for finding and replacing text attributes as well as for text. For example, executing a command such as "replace all selected 12 point Monaco text with 14 point Times text" is possible.

The ability to leave bookmarks within the document is supported. The user can make any selection (text, graphics, or imported data) and "mark" it. We have not yet decided how the user will access these marks. Some possibilities include text markers as in MPW, iconic markers shown along the scrollbar (en-

abling the user to simply click an icon and jump to the marked data), and a sort of "live" table ofcontents in which all marks are listed as text or pictures, and the user may select from among them to go to the marked data.

The user may also traverse his/her document by specifying the page number to jump to. The current page number and number of pages in the document are always displayed in the view.

Linking

Jane provides the capability to link related items in a document, or between documents. To create a link the user specifies its start and end (although once the link is constructed it is bi-directional). Jane understands two different kinds of links, which we will refer to as navigational links and hot links. Navigational links allow the user to select a linked item (at either end) and traverse that link to the other end. In other words, when a link is followed (by double-clicking on a link icon, perhaps), another view opens up showing the opposite end of the link.

Navigational links are used to do bookmarks, automatic endnotes, footnotes, bibliographies, index entries, and annotation notes (and we'll probably think of even more uses). For example, the user selects "Endnote" from a menu, and Jane automatically generates a number or letter to mark the location of the endnote in the text. The user is prompted to enter the text of his/her note in a new window. The note is inserted into the document along with other notes (at the end of the section or chapter, or the end of the document) when the window is closed. The number or letter indicating the note is then linked automatically to its entry on the Notes page. Selecting the note number and following the link opens up the Notes page; likewise, selecting a note on the Notes page and following it causes a view of the document to open up, showing the reference to the note within the text.

Hot links can also be navigated, but in addition the data between the two ends of the link is shared. The user specifies when the data is to be shared. For example, the user may edit the data on one end of the link and then "push" the data to the other end of the link. This is the same operation as deleting the data on the other end of the link, and pasting the modified data in its location.

Hot links are used by Jane to make the editing of imported data much easier than it has been in the past. For example, assume that the user has pasted a Tuffy chart into a Jane document, and set up a link between the original chart in a Tuffy document, and this pasted copy. If the user decides to change a value in the chart, s/he may follow the link from the copy of the chart to the original. This activates Tuffy, and causes the Tuffy document containing the chart to open up. This is far easier than forcing the user to return to the Finder and search for the document in which s/he created the original chart. The user can now edit the chart in the Tuffy window, and then push the data to the other end of the link, so that the modified chart automatically replaces the older version of the chart in the Jane document.

Links are represented by an icon, which could vary in appearance according to the type of link. We may want the icon to clearly differentiate between navigational links and hot links. We may also want to differentiate between further subclasses of links, such as the examples we gave above for navigational links. This an open issue which needs to be solved at a system-wide level. In Jane it is also possible to hide the link icons, if so desired.

The Book Model

Jane supports the idea of a "book". A book is a kind of folder containing files representing various parts of a document. These include the chapters or sections of the document, the table of contents, the index, the bibliography, and the glossary. The advantage of maintaining separate sections is to enable different authors to work on the same document at the same time. In addition, the various parts of the document have knowledge of each other; whenever one section changes in some way that affects the document as a whole, the other sections are notified and updated appropriately. An example of this is automatic page numbering between sections of the document.

Collaboration

The collaboration model supports the idea of different classes of users. Jane recognizes three types of users, which we refer to as authors, editors, and reviewers. Documents may be checked in and out by all three types. There may be multiple authors, editors and reviewers associated with each document. Authors have full editing privileges. Editors are given a copy of the original document to edit as they please, but the contents of the original document are not affected by the editors' modifications (unless the author applies those changes, as described under Markup, below). Reviewers may read the document and attach notes to it. Defining three classes of users helps to insure that documents are not modified in-advertently. For example, a reviewer has no intention of changing the document contents; s/he simply wants to read the document and leave his/her impressions for the author to read in the form of notes.

Notes

Authors, editors, and reviewers can all leave notes attached to a document. This is generally done by selecting some data (text, picture, or imported data of any type) and creating a new note. When a note is created, a new window opens up in which the user can enter the contents of the note. An icon is inserted at the end of the selection to which the note refers. Double-clicking on this icon always opens up the note, if it is not open already. There is no limit to the size of the note.

Notes are saved with information about who created them, and when. There is also an optional subject (title?) field, which must be filled in by the note creator if it is to be used. The user may then open up notes selectively (by double-clicking on their icons), or may look at a group of notes, such as all notes written by one individual, or all notes written on a certain date, or all notes falling under a certain subject heading. The manner in which notes are displayed on the screen or printed is a user preference. The user may choose to display them in the document margin, or in individual views, or altogether in a single notes window. Notes may also be selectively printed,according to author, date, or subject.

Markup

When a document is checked out by an editor, a copy of the document is created for the editor to modify as s/he pleases. When the editor checks the document back in, his/her version is saved along with the original document. The editor's name or some other identifier is saved with the editor's version of the document. When the author then checks out his/her document, s/he may also view the version created by the editor. In addition, the author may request Jane to automatically generate a marked up overlay to be viewed on top of the original document. The marked up overlay indicates changes made by the editor

in his/her version of the document. Deletions are shown by a cross-out line. Insertions of just a few characters are shown above the insertion point; larger insertions are indicated by a note icon, which can be opened to read the insertion. Style changes are highlighted in the editor's highlight color. Each marked up overlay is color-coded. The colors are selected automatically, in order to insure that each editor's color is unique. This enables the author to view multiple overlays at the same time, and still be able to differentiate one editor from another.

The real power of this model lies in enabling the author to selectively apply changes made by the editor to the original document. In other words, the author may select a change mark on an overlay and execute a "Do It" command. The original document is modified in the way indicated by the change mark. Change marks can also be applied more automatically; the author can simply execute a command to make all the indicated changes of a given editor. This results in the editor's copy simply replacing the original document. It is also possible to apply a class of changes automatically, such as all font changes, for example.

Pink "Freebies"

The Pink toolbox provides some powerful capabilities which are available to all applications, and which, of course, Jane will take advantage of. For example, unlimited undo and automatic document versioning are supported by the document architecture model[2]. International support is automatically provided by the layout manager[3]. Jane will also take advantage of scripting[4] in two ways: 1) the system-wide scripting model will be supported in Jane so that the user will be able to create his/her own scripts, and 2) scripting will be used by us as the underlying mechanism for implementing some features of Jane.

It is also likely that other features we have already mentioned above will be implemented on a system-wide level. Certainly hot links and basic navigational links are going to be available across applications, and some kind of annotation capability will be available as well. Since these features are essential to Jane's success, we feel it is important for us to list them in the Jane feature list and to take the first stab at describing a possible interface for them.

---

2. See CHER, section 2.2.2.
3. See Line Layout, section 2.7.3.
4. See Scripting, section 2.2.5.

Project Dependencies

- **The scope of the project needs to be kept within reasonable bounds.** Jane is a very ambitious proposal for a word processor, and it may be that we are hoping to accomplish too much with only three people on the team. If this turns out to be the case, it may be necessary to trim some of the features we hope to offer.

- **Cooperation between the Jane team and other Pink groups.** It is essential to the success of Pink that the Jane team provide feedback on every part of the Pink system which is exercised by Jane. Jane, in turn, cannot succeed without other groups in Pink being responsive to that feedback.

- **Collaboration between the Jane team and the Pink Human Interface team.** The Jane team needs to work closely with the Pink Human Interface team to solve the many human interface issues which will arise during the development of Jane. Some issues will be specific to Jane, and we hope to have the help of the Human Interface team in solving these. We also will need to work closely with them on solving issues that affect all Pink applications, such as hot links and collaboration.

- **Cooperation between the Jane team and the Pink Finder team.** We will need the Finder to help support the idea of a book metaphor.

- **Ability to "plug in" 3rd party spell-checkers.** We are not planning on providing our own spell-checker, as we feel that is best left to 3rd party developers who specialize in such matters. We are assuming that the toolbox will provide the capability for the user to run the spell-checker of their choice concurrently with Jane. This "plug in" mechanism would be used for other features, such as thesauruses and grammar-checkers, as well.

- **Our goal must be to ship.** For Jane to be successful, its goal must be to become a shippable application as opposed to a research project. This is the only way we can prove that a real application can be developed on a Pink platform.

- **The successful completion of the Pink system.** Pink must succeed as a new operating system and toolbox in order for Jane to succeed. For Pink to succeed it must deliver the advanced features described in Big Pink #3, have reasonable performance and run with a minimum of bugs (don't ask us how many "a minimum" is).

Current Status

We have written a prototype demonstrating some aspects of Jane using MacApp. Most importantly, this prototype demonstrates direct manipulation text. A selection of text may be grabbed (by a handle provided on the left side of the selection) and dragged to a new location within thetext. This prototype has been user-tested to determine the feasability of direct manipulation text. Based on the results of the user-testing, we have decided to go ahead with the plan to implement direct manipulation text, with some minor changes to the interface.

We have written a very simple prototype in Pink which allows the user to draw shapes and enter text into them. The most interesting thing about this prototype is that it uses CHER, the document architecture model. Thanks to CHER, different views of the same document can be displayed at the same time, such that a change (for example, a newly drawn shape) which is made in one view causes all of the document views to be updated to reflect the new information. Unlimited undo/redo is also supported by the version of CHER which we are using.

Currently we are in the process of defining the Jane feature list and designing the user interface. We are creating mini-prototypes using Supercard for this purpose. We hope to have complete specifications for Jane ready within another couple of months. An updated ERS will be issued at that time.

Tuffy

Pink Graphing Application

56

By Dave Anderson and Cindy Frost

56

Introduction

This paper is a proposal for a data graphing application to be developed on the Pink system. Many of the ideas embodied in this paper originated from an earlier proposal for Tuffy written by Amy Goldsmith. We have attempted to integrate her ideas with our own when writing this proposal.

Tuffy was inspired by Edward R. Tufte's book The Visual Display of Quantitative Information [Tufte 83]. In his book, Tufte discusses the power of data graphics to reveal information and relationships in data that would otherwise go unnoticed by the casual observer. He presents examples of many excellent graphics, along with many graphics that provide either inadequate and /or inaccurate presentations of data.

Tufte cites several sources for the lack of graphical sophistication and integrity prevalent in data graphics today. These include the lack of quantitative skills in graphic artists, and attitudes prevalent among producers of quantitative graphics that statistical data are boring and that graphics are only for the unsophisticated reader. Although Tuffy cannot necessarily change attitudes, we believe that there is tremendous potential for overcoming the first problem with an application that brings the power to create interesting and sophisticated graphics to the individuals with the understanding of the data. Tuffy is just such an application.

In his book, Tufte presents a theory on data graph design which embodies a set of guidelines for generating high quality data graphics. Included in the theory are a set of quantitative measures for a graphic's adherence to the guidelines. These measures are a key part of what Tuffy will provide the user over other graphing applications. With these measures as feedback, users should be able to produce superior data graphics.

One might ask what Tuffy can provide which is not already provided by the multitude of spreadsheet, statistics and graphing applications currently in existence for the Mac. In reviewing the existing applications, we discovered that even the most general products, like CricketGraph, KaleidoGraph and spreadsheets, place significant limitations on the size of the data sets and the type of data which can be plotted (e.g. most spreadsheets allow only categories to be plotted on the X axis, treating even numerical values strictly as category labels). A few provide for data of more than 2 dimensions, but usually only through a Z variable in 3D plots. Although many provide sophisticated control of graph axes and grids, the control is provided through cluttered dialog boxes with little or no visual feedback. Tuffy can provide more flexible and easy-to-use versions of current functionality along with a unique emphasis on data graphic design, empowering users with little or no artistic skill to create sophisticated, accurate and aesthetically pleasing graphics.

Goals

The primary goal of any graphing application is to allow its users to produce alternative presentations of quantitative information in the form of graphs and charts. Tuffy goes beyond this by attempting to aid the user in producing graphs and charts which conform to Tufte's principles for high quality data graphics, and in choosing the optimum presentation for a particular set of data. Tuffy will attempt to determine reasonable alternatives for presenting the data and present those to the user. The user must be able to quickly and easily compare the various presentations to determine which is most suitable for the data. Ideally, the user will discover new ways of presenting the data.

Tuffy will not limit the user to a predefined set of graph formats, but will provide the user with a graph editing tool, allowing the creation of customized data graphics to best present a given data set. These tools must be easy enough to use that even users with little or no artistic skill can create graphics which are compelling, and which provide accurate, revealing presentations of the data.

Tuffy's main emphasis is on the presentation of quantitative information. Tuffy should not be concerned with the acquisition or manipulation of data but should be dedicated solely to the generation of clear and understandable data graphics. By maintaining a focus on the presentation of the data, we believe Tuffy can provide a complete set of tools to help ensure the highest quality data graphics. While Tuffy should ensure that the presentation accurately reflects the data, it should not be burdened with the accuracy of the data itself. This can be better left to the applications with intimate knowledge of the data being dealt with.

Our charter is not only to create a great application but also to help shape the Pink system and provide verification that it is a viable environment for applications. Because of this, we are pursuing design and implementation of code in areas which are well defined even though there are significant areas such as human interface which still require much definition. We believe it is in the best interest of the Pink project as a whole to take this approach, providing early feedback to Pink, rather than completing an entire human interface design before beginning any implementation.

An Application Philosophy

It is our contention that the user can best be served through a set of specialized applications or tools, each excelling at a particular task that the user must accomplish. Although developed independently, these applications are designed to interact closely with one another appearing to the user as an integrated whole, significantly more powerful than the sum of the parts. In Tuffy and most other significant applications, there are major areas of functionality which we believe can be split into "lightweight" applications or tools which can be easily pulled together through the data they act on.

The first and most obvious place to apply this philosophy in Tuffy is to separate the data entry and data graphing capabilities of the product. The data entry application will allow the user to enter data in a spreadsheet style format while the graphing application will focus on presentation of that data. The two will interact by sharing the data document model between them. Providing the data entry capabilities as a separate application will maintain the focus of Tuffy on data presentation and will allow the replacement of the data entry application with more sophisticated spreadsheet or statistics applications in the future. It will also provide the added benefit of imposing a design which incorporates adequate support for data sharing between applications, proving the validity of the concept for Tuffy, and Pink applications in general.

We also intend to follow this philosophy when developing the graphing capabilities of Tuffy. For example, graph format editing capabilities could be viewed as a tool separate from the the tool which manages multiple graph views on a document. Again the interaction between the two comes from sharing a common document model. In this case, the graph format description model. Users will be allowed to work with the graphing tools which are of interest to them without additional clutter produced by unused tools. This split will also allow for the easy replacement of individual tools, such as replacing a simplistic drawing tool with a more powerful one.

This is the direction we intend to take with Tuffy and believe it is a direction worth considering for all Pink applications. The realization of such a philosophy requires powerful tools for the

sharing of information between applications. We are relying on the Pink Toolbox to supply such tools.

Capabilities

In this section we present the capabilities which Tuffy must provide in order to accomplish the above stated goals. Tuffy must provide support for producing a set of standard data graphics as well as custom graphics defined by the user. At all times Tuffy must provide adequate feedback on the quality of the graphic being created. Tuffy will allow multiple simultaneous views on the same data to allow easy comparison of the different graph formats, aiding the user in choosing the format which best presents the data.

Graphs will be composed from a set of graphing elements which include data plots, data points, axes, grids, legends, text fields, etc. Tuffy will provide standard elements for each of the element types and will also support user definition of new elements. Direct manipulation of graph elements will be an important feature of Tuffy, allowing users to quickly and easily modify both standard and custom graph views.

Document Links

All of the data presented by Tuffy will be provided by other applications (although the application may have been developed specifically to provide data to Tuffy). Therefore, the document links provided by the CHER component of Pink will be crucial to the success of the project. Tuffy will need to rapidly push and pull data across the links to support the direct manipulation of data points.

Graph Views

A graph view provides the user with a view of a particular data set in a particular format. Graph views are fully functional windows which may be individually resized and positioned. Multiple views of the same data may be displayed simultaneously. Unlike some applications, creating a new view does not create a new copy of the data, but rather just another view on the data. Changes in the data are reflected automatically in the view. Opening multiple views will be a very quick and simple task allowing the user to easily compare different graphs of the same data. Providing a good metaphor for handling multiple views on the data will be important. It must be made clear to the user where the real data actually exists(e.g the difference between and document window and a view window must be clear).

View Control

Graph size and view size are independent. The Graph may extend horizontally and vertically beyond the extent of the view with support for scrolling. The view may be zoomed in for detailed work and zoomed out to gain perspective on a large graph.

View Composition

Graph views are composed of a collection of graphing elements. A simple line graph, for example, could be composed of a line plot, an X axis, Y axis, legend and text labels. A single view may contain multiple instances of a single element type. For example, several Y variables could be plotted in a line graph simply by including an individual Y plot for each variable in the graph. Additionally, an individual Y axis for each variable could be included to allow different ranges for each variable (similar to the double-Y plots found in some packages). Additional text elements could be added, etc.

View Gallery

A gallery of standard graph views will be provided. These will include, at a minimum, 2D scatter, line, area, bar, column and pie and 3D scatter, line, column and bar. In addition new custom views may be created by the user and added to the gallery of standard views. The gallery could be presented to the user as a set of icons which are actually miniature views on the data selected for plotting. With Tuffy, the user has the option to ignore the standard views and create one of his own from the graphing elements provided. The custom graph types can be saved as stationery and distributed as such. This would allow a journal publisher to distribute graph formats to authors to be used in presenting data in an article.

Direct Manipulation

The graph views will provide direct manipulation of data points. Ideally, changes in a data point's value would be reflected in all views containing that data in real-time, performance allowing. In any case, any change in a data point's value will be reflected in all views containing that point after the value has been assigned, including views on the same data in other applications. For very dense data, or for graphs like line-plots which do not plot individual data points, a magnifying glass cursor will be provided to allow the user to more easily select the desired data point. Additionally, this magnified view could display the numeric values associated with a selected data point, making direct manipulation of the data even more powerful.

Direct manipulation will not be limited to data points alone, but will be provided for all of the graphing elements for which it makes sense. For example, rather then requiring the user to enter ranges for a graph's axes in a dialog box, the user could be allowed to drag the axes' endpoints to the desired locations. Of course the user will still be able to enter any numerical values directly where high accuracy is a requirement

Backgrounds

Graph backgrounds may be any graphic the user desires and may in fact be multiple graphics overlayed. A typical background would be a map over which various data points or paths may be plotted.

Data Plots

Tuffy will provide all of the standard types of plots found in existing applications along with several new ones unique to Tuffy. Basic 2D plots types include scatter, line, area, bar, column, area and box-plots. Additional 2D plots will exist for plotting additional dimensions to the data.

These include intensity varied and labeled plot points. As many plots as desired may be overlayed on a single graph simply by including additional plot elements for the graph.

Tuffy will take advantage of the 3D graphics capabilities provided by Albert, to provide 3D scatter, line and column graphs. Allowing various viewpoints and perspectives on the 3D graphs should be made straight-forward by Albert. Good methods will need to be developed for user manipulation of 3D views. The methods developed should be common to all Pink applications. A likely candidate is Michael Chen's virtual sphere controller.

NOTE: We need to consider plots of both discrete and continuous data. Plottng continuous functions may provide support for the curve-fitting functions found in other applications today.

Plotting Elements

Individual data points may be plotted with a variety of plotting elements. Standard elements include circles, squares, diamonds, text, lines, bars, pie wedges, etc. 3D objects such are spheres and cubes will also be supported. These data elements will be able to plot additional dimensions of the data by allowing data values or ranges of values to control their shade, color or shape. Tuffy will make very limited use of patterns in order to avoid the moiré vibration associated with many patterns. Tuffy will also support import of objects from other programs to be used as plotting elements.

Tuffy will provide an interface for the user to view the complete data associated with a selected data point.

Axes

Graph axes are composed of several individual graphing elements; a line, tick marks, grids, and labels. Each of the elements comprising an axis may be individually controlled. Direct manipulation of these elements will be provided wherever possible.

Axis Line

The axis line defines the range of the axis. The axis' range and the range of the graph dimension it represents are independent. (e.g. if the total X domain of a graph is (0, 100) the axis range could be limited to the actual range of the data points, say (6 ,88)). Limiting the length of the axis line to the range for the data provides additional information about the data (minimum and maximum values), thereby improving the data-ink ratio. See p149[Tufte 83].

Tick Marks

Axis tick marks may be any of the elements used for plotting data points. Tuffy will provide major and minor tick marks with major tick marks tied to the axis labels. The use of the plotting elements for tick marks will allow additional dimensions of the data to be presented as variations in tick marks are well as data points. An example of this can be seen on p44[Tufte 83].

Tick marks may be spaced at even intervals, logarithmic intervals, or they can follow the actual data points. Tick marks defined as the latter, along with the lack of an axis line, would create the distribution axis of Tufte's dash-dot-dash plot on p133 [Tufte 83].

Labels

Axis labels will be standard text objects which will allow for wrapping of individual labels. Their range and increment may be generated automatically by Tuffy or specified by the user.

Grids

Grids provide the same position options associated with axis tick marks allowing spacing at even or logarithmic intervals, or aligned with data points. Additionally, grid lines may be individually placed by the user to indicate significant data values (e.g. historic events on a time axis).

Text

Tuffy will support text elements to be included with graphs. Multiple fonts, sizes and styles will be supported. Additionally, complex paths for text may be defined, allowing text to follow a line plot for example. Text elements provided automatically by Tuffy (e.g. labels, legends) are identical to those created by the user with the drawing tools and may be manipulated as such.

Tables

Tuffy will provide tools for creating tables of data. Tables are especially useful for very sparse data sets (indicated by a low data-density ratio) or when presenting exact numerical values. Tuffy could allow the creation of table templates into which data sets could be inserted. All the text styles support by the Pink toolbox will be available for tables.

Statistics

We are currently investigating which of the capabilities provided by current statistics applications could be provided by a separate Pink statistics application (or perhaps as part of a spreadsheet or other program), and which capabilities are so closely tied to the plotting of the data that they must be contained in Tuffy itself. For example, can curve fitting be supported by Tuffy without building in specific curve-fit functions? We will probably at least need to provide support for error bars and the indication of mean value and standard deviation for data sets.

Drawing Tools

A minimal set of drawing tools will be included to allow the user to add text and basic 2D and 3D geometric shapes to the data graphics. Complex drawings can be imported from more powerful drawing applications to be used as backgrounds, data points or adornments to the graph.

Graphing Tools

A set of graphing tools will be included to allow the user to select, add, delete and manipulate the graphing elements provided by Tuffy. This includes direct manipulation of the data points themselves.

Graphic Databases (maps)

Tuffy will support graphic databases which can define collections of graphical objects for which data values may be used to control the shade, shape or color of each object. An example of a graphic for which this applies is on pp17-19[Tufte 83]. This graphic presents cancer rates for all counties in the United States as gray shades for the counties on a U.S. map.

Small Multiples and Animation

Small multiples are a series of graphics, resembling the frames of a movie, showing the same combination of variables indexed by change in another variable, typically time. Tuffy will provide support for easy layout of small multiples on the output page. In addition, Tuffy will allow animation of a graphic by providing real-time manipulation of the indexing variable while viewing its effect on the plot. Additionally, investigation into the export of animation sequences to other Pink applications (e.g. a presentation application) is warranted. This is a capability which is probably desirable in other applications and needs to be resolved in a general manner for Pink.

Printing

Tuffy will support printing of multiple graphs on an output page. This will include some minimal page layout capabilities to allow the user to size and place the available graphs on the page.

User Feedback

Tuffy will provide feedback to the user on the quality of the graphic being created. A small status window will be available at all times with more detailed information available at the users request. The measures are those presented by Tufte in his "Theory on Data Graphics". It would be desirable to present the measures for each of a group of graphs to allow the user to easily compare between them. Perhaps

Data-ink ratio

This is the simple ratio of ink used to represent data to the total ink used to print the graphic. It can be used to aid users in following Tufte's principle of data-ink maximization. Tufte suggests that maximizing the data ink ratio is beneficial to all quantitative graphics. Tuffy will provide the user with the value of the data ink ratio for a graphic as well as ranges which are considered acceptable. Additionally Tuffy will highlight redundant data-ink and non data-ink to make it easier for the user to identify areas of the graphic which may be removed without loss of infor-

mation.

Data Density

The data density is the number of data points per unit area in a graphic. Tufte suggests that many graphics are way too sparse and can be shrunk considerably to improve the data density, leaving room for other information on the page.Tuffy will provide this ratio for a graphic along with ranges indicating whether graphics are too sparse, acceptable or too dense. Tuffy could also suggest a size for the graphic which provides an optimal data density.

Graphic Proportions

This is the simple ratio of the graphics width to its height. An ancient but still practical rule of graphic proportions is that of the golden rectangle. The width:height ratio for the golden rectangle is approximately 1.618:1. Graphics which approach these proportions have been found to be more aesthetically pleasing. Tuffy will provide the proportions for a graphic along with ranges of acceptability.

Human Interface

In this section we describe the user-interface elements which will be included in Tuffy which are necessary to provide the capabilities described above. Currently this section is primarily an outline, listing the elements which we have identified with some detailed descriptions in areas in which we have begun study. We plan for this section to evolve as the application design evolves.

Numeric Data entry

Numeric data entry will be supported through a spreadsheet like interface. The user will be able to create columns of data which can be plotted by the graphing application. The following features will be supported:

Addition of new empty columns
Editing of column labels
Entry data values and categories text directly into spreadsheet cells
Specify units and display formats for column data. This will likely be accomplish through a dialog box brought up by perhaps double-clicking the column header.
Control the display width of columns by dragging column separator lines.
Selection of columns
Multi-cell selection including noncontiguous cell
Delete, cut, copy and paste of cell selections
Hypertext links between cells
Eternal undo of commands
Warm links to data in other application documents

Variable and Graph Format Selection

Select variables for plotting (from spreadsheet or other graph)

Select graph format for plotting from gallery
Associate variables with plots
Determination of graph format alternatives and the presentation of those alternatives
Data Variable/Graph Variable associations (choosing variables for X, Y. etc}

Graph Format Determination from Document Data

Tuffy will attempt to determine reasonable graph formats for the selected document data. Some of the criteria which will be used to make this determination are as follows:

Variable count
Variable range
Data point count
Variable type or units

Graph Views

Tuffy will support multiple simultaneous views on a single document. In order to avoid the potential screen clutter and confusion caused by a proliferation of windows we need a mechanism for collecting and managing document views. Our current thinking is based on a design which has a single document window with iconic representations of each of the graph formats selected for that document, perhaps with the currently selected "favorite" format displayed in the window. The icons can be double-clicked to open a graph into a window, allowing manipulation and comparison of the graphs. The windows will appear different from standard document windows to indicate to the user that the opening and closing of these views is a much lighter weight operation than that of opening or closing a document.

The graph views when open will act as standard Pink windows supporting normal sizing, zooming and scrolling operations. Various tools can be used to manipulate the format of the graphs.

Graph quality measurements

Format for the presentation of Tufte graph quality measures
identification of non-data ink for the user
Presentation of suggested changes to graph

Graph Format Editing and Manipulation

- Axis control
  - Select axis type (log, semi-log, distribution, quartile)
  - Set ranges
  - Set increments
  - Select element for tick marks
  - Attach variable to tick mark elements
  - Select grid types
  - Attach to specific plot
- Plotting element control
  - Marker style selection
  - Marker color selection
  - Attach variables to plotting element attributes, e.g. shade or color
  - Specification of overlay handling
- Direct manipulation of data

View detail information on data point
Adjust data point directly
Add new data points
Manipulation of other graph elements such as those created by the user

Graph Format Creation

Standard and Custom graph element parts bins (similar to Constructor in HOOPS)
Adding elements to graph
Connecting elements such as plots and axes
Customizing elements and saving in custom parts bin
Creation of stationery to allow reuse of graph formats

Printing/Saving

Place and size graphs on the output page
Save document with working graph set

Issues

Where will the data sets come from? In other words will an application exist which will provide data to Tuffy or will Tuffy need to provide this application?

We need to determine whether several smaller applications can be integrated well enough to work as though they were one monolithic application.

Can we take advantage of Bjorn's constraint work? A couple of uses include constraining X axis labels to align vertically with their corresponding tick marks, and controlling the way in which overlapping data points are plotted.

We have attempted to review as many of the existing graphing applications as possible. However, there are still others which we intend to look at which may prompt new features or changes to those already proposed. These include Wavemetrics' Igor, and several mainframe graphing packages.

Due to the finite resources available for this project we need to determine how much of the capabilities must be included to create a "compelling" application.

Bibliography

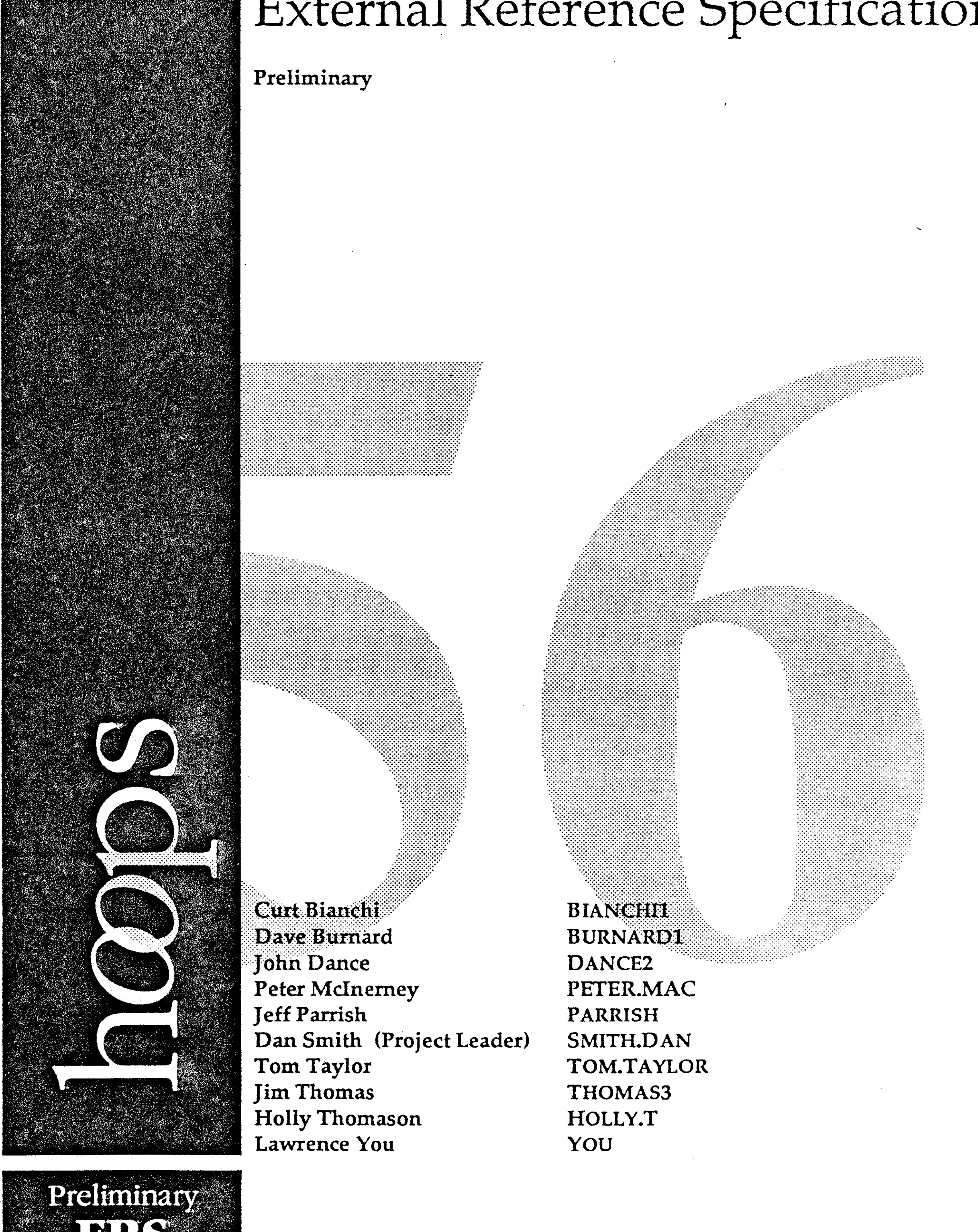
Tufte 83 Tufte, Edward R., "The Visual Display of Quantitative Information", Graphics Press, Connecticut, 1983.

A Human-Oriented Object Programming System

56

External Reference Specification

Preliminary



Curt Bianchi	BIANCHI1
Dave Burnard	BURNARD1
John Dance	DANCE2
Peter McInerney	PETER.MAC
Jeff Parrish	PARRISH
Dan Smith (Project Leader)	SMITH.DAN
Tom Taylor	TOM.TAYLOR
Jim Thomas	THOMAS3
Holly Thomason	HOLLY.T
Lawrence You	YOU

Preliminary
ERS

Apple Confidential

Table of Contents

Motivation 3.5-1
What is hoops? 3.5-1
Why hoops? 3.5-3
Design Principles 3.5-6
What hoops is not 3.5-8

The Foundation 3.5-9
Representation of Programs 3.5-9
Presentation of Programs 3.5-15
The hoops Object-Oriented Framework 3.5-17

The hoops Human Interface 3.5-20
Basic Principles 3.5-20
Human Interface Elements 3.5-21
A hoops Viewer Sampler 3.5-23
Dynamic Browsers 3.5-32
Source Code Editing 3.5-34
Navigation 3.5-38
User Scenario 3.5-40

Program Management 3.5-43
Projects 3.5-43
Stationery 3.5-44
Publishing 3.5-45
Conditionals 3.5-46
Configurations 3.5-49
The Build Process 3.5-51
Importing and Exporting 3.5-53
Collaboration 3.5-54

Debugging 3.5-56

Constructor 3.5-61
The Foundation 3.5-62
The Constructor Human Interface 3.5-68
Constructor Details 3.5-74

Dependencies on Pink 3.5-81

Dependencies on CompTech 3.5-84

Glossary 3.5-85

Color Figures 3.5-89

56

Motivation

What is hoops?

Tomorrow is the most important thing in life. Comes into us at midnight very clean. It's perfect when it arrives and puts itself in our hands. It hopes we've learned something from yesterday.

- John Wayne

Pink Programming Environment

hoops is a new programming environment for the Pink Macintosh. It is designed from scratch to be an integrated extension of the Pink toolbox rather than a collection of tools added after the event. hoops is intended to provide the same advance in usability for the programmer that the Macintosh provided for the application user.

hoops is a Pink application in the fullest sense of the word. It makes the widest possible use of standard Pink services, and has a fully Pink human interface.

Architecture

hoops is not just a specific programming environment. hoops is designed to be adaptable and extensible. This is achieved by building hoops around an architecture that is powerful enough to grow and change through the next decade. This means that all the needs of potential hoops users will not be accommodated in the first version, but the potential for addressing these needs is part of the design.

Features

- Graphical, direct manipulation interface
- Multiple program views and organizations
- Language-sensitive editing and navigation
- Incremental compilation and linking
- Automatic program build facility
- Integrated symbolic debugging
- Integrated team programming support
- Integrated interface building
- On-line documentation
- Multiple fonts, styles, graphics

Initial Clients

The primary clients of hoops are Apple internal system developers and Pink application developers. Because of the object-oriented nature of the Pink toolbox, the needs of these groups are in many cases closer than in the past. Secondary clients include any other internal or external programmers developing for the Pink environment.

Host Architectures

hoops is being built upon, and tightly integrated with, the Pink system. It is designed to be portable to different hardware platforms running the Pink system software.

Hardware Requirements

In addition to the Pink minimum hardware configuration, hoops is expected to require no less than:

- Memory - 4 MB.
- Disk space - 40 MB.
- Screen size - 12" diagonal.

Why hoops?

Strategic Significance

...a winning strategy is like a pound balanced against an ounce..
- Sun Tzu

The purpose of hoops *and* Pink is to ensure that the next generation of innovative software gets developed on Apple computers, by providing a platform so attractive that developers will choose us instead of our competition. Pink provides the foundation for innovative software; hoops provides the tools to facilitate its creation. Both are needed to create a programming environment that will attract the developers needed for our success.

For some five years Apple has enjoyed a significant strategic advantage: the Macintosh has been the computer of choice for the development of innovative personal computer software. The reason for this is that developers were attracted to the Macintosh. It was both fun and challenging to program. It had a superior human interface and programming toolkit, both of which enabled the expression of new kinds of software. In short, developers *wanted* to write software for it. Lacking a comparable human interface or toolkit, our competitors have been in the position of catching up to us for the last five years. Well, they're catching up, and in some cases overtaking us. Pink and hoops are designed to widen the gap again.

Ultimately our computers sell because of great programs written by third party developers. Without these developers we are just suppliers of expensive toys (no matter how good our system software is). By attracting and empowering these developers we are making it possible to sell hardware.

Limitations of Present Tools

Give us the tools, and we will finish the job.
- Winston Churchill

Why do we need an entirely new development environment? For essentially the same reason we need a new Pink software platform. The size and complexity of current Macintosh applications is reaching the limits of both the toolbox and the development tools. Pink addresses the limitations of the toolbox; hoops addresses the limitations of the development tools.

A basic premise behind the Macintosh toolbox was that, by supplying a powerful integrated library, we would provide our developers with the leverage necessary to create great applications. This principle was further extended by MacApp, and will be carried to unprecedented lengths by the Pink toolbox. There is, however, a potential downside in this wealth of available power. In order for developers to use the appropriate pieces of Pink toolbox functionality, they must be able to find them and easily learn how to use them.

In fact, programming a Macintosh has always been considered a difficult task, compared with the more traditional environments on our competitor's machines. Part of the difficulty has been the amount of information that is needed before you start. *Inside Macintosh* has many chapters, all of which seem to assume that all other chapters have been read. For a printed document, *Inside*

Macintosh is mostly very good. It is the printed medium itself that is inadequate to the task.

Many other programmer tools are based on program representations that are essentially electronic paper. These representations inherit many of the problems of their printed ancestors, including poor cross-referencing and navigation aids. hoops must provide much more powerful and direct data access because Pink will be an order of magnitude larger than the original Macintosh. hoops addresses these needs by providing tools to manage and navigate all the data associated with a programming project, including documentation, source and resources. Furthermore, object-oriented programming imposes unique representational and navigational demands on a development environment. hoops is designed to meet those demands.

Besides being weak in program representation and navigation, traditional tools can be slow and obtrusive. Because they are based on batch processing they are generally poor at interactive tasks. Long program build times encourage acceptance of inferior solutions, rather than experimentation and fine tuning. Programmers are forced to attempt to solve many problems at once, rather than one at a time. They are continually disrupted by long idle periods, rather than being allowed continuous concentration on a problem. hoops addresses these issues by taking an incremental and interactive approach.

No matter how good our current programmer tools are in their own terms, or how much effort we put into improving them, there are difficult limitations inherent in the plain-text, batch-mode model of programming.

Programming for Humans

To expect a man to retain everything that he has ever read is like expecting him to carry about in his body everything he has ever eaten...

- Arthur Schopenhauer

hoops is designed with the needs of the human uppermost.[1] Where possible and appropriate, hoops hides and automates everyday bookkeeping, while providing powerful and flexible tools for understanding and writing programs. hoops aims to increase individual programmer productivity, by applying the same human interface principles, of direct manipulation rather than textual commands, and recognition rather than recall, that have served us so well for our other end users.

The needs of the individual are placed very high in the order of implementation of hoops, because if we do not attract individuals, and enable them to learn the new Pink system, we will almost certainly fail to attract larger groups. When Pink first ships, Apple will be in a position much like when the Macintosh first appeared. Almost none of our outside developers will have

1. This is in contrast to traditional environments which often have ways of performing actions that are more for the convenience of the tools in the environment than for the programmer, e.g. ordering program elements in such a way as to allow single pass compilers. This was often necessary because these methods evolved in times when the resources available for the individual programmer were much poorer.

knowledge of, or experience with this totally new system, and one of our first priorities must be to attract and educate them. hoops will play an important part in the success of this approach.

hoops provides the means to make the Pink toolbox accessible and comprehensible, and allows programmers to apply the same organizational techniques to their own programs. Developers will benefit because programs of the complexity of current applications will be more easily written, understood, and maintained. Also, hoops will make possible programs whose creation is beyond the capabilities of existing tools.

As a program gets large, the development environment can significantly impact an individual's ability to deal with it. Innovative software comes from innovative *individuals*, even when these individuals are working in teams. We need to provide these individuals with the ability to express and grasp large and complex programs.

Testing the Pink Architecture

We did consider a full scale test, but...
- Titanic designer

Building applications, and using them, is the only true test of a new software architecture. Producing a development system will not only add to the application test base for Pink, but will push the system in many ways beyond those of typical applications.

Design Principles

Conceptual Simplicity

Our life is frittered away by detail... Simplify, simplify
- Henry David Thoreau

Rather than appearing as a disparate collection of tools tacked on after the event, hoops will be a natural extension of the Pink system and an exemplary Pink application. hoops unifies and simplifies program components, while presenting a coherent unified face.

Efficiency

The world is a dirty place but I wouldn't want to dust it.
- apologies to Steven Wright

hoops will be efficient in the way it helps programmers. Actions that are common will be especially easy to carry out, and wherever possible will not result in disruptive processing delays.

Unobtrusiveness

We never do anything well till we cease to think about the manner of doing it.
- William Hazlitt

An unobtrusive environment is one that doesn't intrude on an individual's thought processes. A key element of this is rapid turn-around of program changes. hoops will provide incremental compilation and linking – most changes will be ready for testing within seconds. In addition, hoops will be well integrated with the rest of Pink. Our goal is to make hoops feel like part of Pink, rather than a completely separate environment.

Direct Manipulation

Remember: No matter where you go, there you are.
- Buckaroo Banzai

Where possible hoops uses direct manipulation. The aim is to make the various components of a program concrete and explicit objects, rather than abstract and implicit representations.

Recognition Versus Recall

Memory is the thing you forget with.
- Alexander Chase

Humans can recognize a far greater body of knowledge than they can directly recall. This is seen in the fact that most of us have much larger reading than writing vocabularies. hoops will provide electronic aids to shift the burden of recall from the programmer to the environment wherever possible.

Extensibility

One never notices what has been done; one can only see what remains to be done...
- Marie Curie

hoops will be built with extensibility always in mind. hoops will be a multilingual environment, so wherever possible hoops will provide general purpose

mechanisms that can be specialized as required. We can expect to get some things wrong and miss some functionality altogether, so we need to make it possible to correct our mistakes and repair our oversights. In addition, we want to make it possible for developers to exercise their ingenuity by enhancing hoops. The key to extensibility lies in an object-oriented design and implementation. Thus, modifying or extending hoops will involve exactly the same process that is used to create Pink applications. The object-oriented design will permit extensions ranging from simple behavior modifications to the addition of new tools and support for new languages.

What hoops is not

It isn't MPW or UNIX

Change is not made without inconvenience, even from worse to better.
- Richard Hooker

hoops is not a port of MPW or UNIX development tools. The way programs are represented and manipulated in hoops differs fundamentally from these traditional environments. In order to provide an increase in representational power and to support an interactive, incremental hoops environment, we have moved away from the view of program source as undifferentiated text in a collection of files. This means that many traditional ways of doing things have been replaced or modified. In many cases, the old mechanisms are redundant or supplanted by mechanisms that are more intelligent about the structure of a program. As might be expected with such a paradigm shift, not all changes will be seen uniformly as improvements. We intend to keep these situations to a minimum but expect some adjustments will be necessary.

It isn't All Things to All People

Initially anyway, hoops is designed to enable the most productive creation of Pink programs. Later versions of hoops may be employed as cross development platforms, but these are not our initial targets.

hoops is tailored more for object-oriented than procedural programming. While we expect to provide tools that are useful in both programming styles, we expect object-oriented tools to be adapted to procedural uses rather than vice-versa.

hoops is aimed primarily at the experienced programmer interested in developing production quality software. While the system will be approachable to new users, it is not being specifically designed for teaching software development. We hope that hoops will be attractive to the academic programming community, but they are not primary targets.

It isn't Slow

We are dedicated to reducing programmer dead time. This means that we will do everything in our power to make sure that hoops is perceived by the programmer to be fast. The key to maintaining this perception is responsiveness, which is achieved in hoops by incremental processing tools sharing a common pool of accumulated data.

The Foundation

> **foun•da•tion** *n.* **1.** that on which something is founded. **2.** the basis or ground of anything. **3.** the natural or prepared ground or base on which some structure rests. **4.** the lowest division of a building, wall, or the like, usually of masonry and partly or wholly below the surface of the ground. –**Syn. 3.** See **base**.

The foundation of hoops consists of three parts, all of which provide an architecture from which a diverse set of concrete features can be produced. The first part of the foundation is a representation of the semantic elements of programs. The second part is a framework for producing different kinds of views, or presentations of a program. The last part is the implementation of hoops as a set of classes, much like Pink itself. Not only do these parts permit us to build the kind of system we would like, but more importantly, they provide an architecture that permits hoops to be extended in ways *users* would like.

Representation of Programs

A thing is an abstraction even if there are no concrete examples of it.
-Anonymous

At the heart of hoops is the ability to understand and manipulate the semantic elements of programs. hoops directly models the structure of programs in terms of classes, functions, variables, and other semantic entities. This is in contrast to conventional development systems, including MPW, which model programs as a set of ASCII text streams.

Components

hoops programs consist of *components*. Components are named objects that represent semantic elements of a program. Components fall into three broad categories: *code components*, *organizational components*, and *resource components*.[2] Code components represent those parts of a program written in a programming language. Organizational components are used to arrange other components into user-defined groupings, so as to organize a large number of components. Resource components represent archived instances of objects, such as windows, menus, dialogs, and so on. A full description of resource components can be found in the "Constructor" chapter of this document. The others will be fully described shortly.

2. We use "resource" for lack of a better term, so don't read too much into it. Any suggestions for another term?

Properties

Components have *properties*. Properties are the attributes or characteristics of components. Examples of properties include source code, object code, and descriptions (documentation). A property can be *intrinsic* or *derived*. An intrinsic property is one whose value is stored as part of a component. A derived property is one that can be derived or created from the intrinsic properties of the component itself, or from the properties of other components.

All components, regardless of type, have at least four properties: a description, a set of clients, a set of references, and a container. The description of a component is represented as a Pink text object and therefore capable of including non-textual data as permitted by the Pink text system. The clients are a set of references to the *users* of a component. The references are a set of references to those components that are *used by* this component. The container is a reference to the component's containing component.

Extensibility

An important aspect of hoops is the ability to add new component and property types. This permits hoops to be extended to more fully support the semantics of programming languages as they are added to hoops. To support the addition of new component types, hoops defines components in terms of classes from which new components can be derived. Likewise, properties are represented by classes as well. Furthermore, hoops defines a protocol, or method of communication between *all* components and hoops itself that permits the addition of new components. hoops relies on the Pink dynamic class mechanism to install new components into hoops without modifying hoops itself. To permit the addition of new properties, components have the ability to return the value of "named" properties. To ensure the best possible performance for properties known to hoops (e.g., source and object code) there will be specific access methods for these built-in properties, though they could also be obtained by the named property technique if desired.

Versioning

Components form a natural basis for versioning. Ideally hoops maintains enough information to recreate every version of every property of every component. However, given the limitations of disk space, the user will have to control the amount of information maintained, by archiving or discarding changes beyond a particular date. Furthermore, it may be possible to choose the properties for which version information is maintained.

Code Components

As stated above, code components represent those parts of a program that have source code and possibly object code. Code components are considered to be either *atomic* or a *collection*. Atomic components are the smallest program ele-

ments understood by h*oo*ps, and are indivisible. Collection components provide a means of grouping atomic components together, thus treating the group as a whole. Many languages have such a grouping construct; Apple Pascal has the unit, and Modula-2 has the module.

A representative set of the kinds of code components expected to ship with h*oo*ps is shown below:

Atomic	Collection
constants	classes
type definitions	modules
variables	
routines	
macros	
files	

Atomic Code Components

The purpose of the atomic components should be obvious, as they map to the typical constructs of many procedural programming languages. However, as additional programming languages are added to h*oo*ps it may be desirable to provide specialized components to better support the language. For example, we may wish to define an "operator" component as a special case of the "routine," in order to better support C++ operator overloading.

Note that hoops considers routines to be atomic. The implication of this is that routines are represented as single, indivisible components in hoops. Thus, variables, types, etc. defined locally within a routine are *not* themselves represented as components. For C++, this means that constants, types, and variables defined within a local scope are *not* represented as components.[3] For Pascal, this also means that nested procedures or functions are not components themselves, but are simply part of the enclosing routine visible at the global level.

Collection Code Components

There are two kinds of collection components: *classes* and *modules*. All atomic code components (except the file) reside in either a class or a module. A class contains only those components defined to be part of that class (i.e., its data members and member functions). A module contains a set of related, non-object-oriented code. Like the members of a class, the components of a module can be designated as public or private with respect to the module. This leads to an interface for the module, much like one would create with a header file in a conventional environment. As was the case for atomic components, it may be useful to define additional kinds of collection components as new languages

3. For a definition of C++ scoping terminology, see Section 3.9 of C++ *Primer* by Stanley B. Lippman.

are added to hoops. For example, a unit component might be added to more fully support Apple Pascal.

The File Component

The File component deserves additional explanation. It represents a stream of text, like an MPW text file. It is atomic and therefore indivisible. Like other atomic components, hoops does not maintain any knowledge of the contents of a file.[4]

The basis of components, of course, is to do away with text files as the means of representing programs. Both the class and module component effectively remove their components from the lexical realm of a text file. While this certainly has many advantages (the treatment of a class as a separable thing for one; the automatic generation of interfaces for another), there are some programming constructs for which files may be convenient.[5] Furthermore, file components may be useful to add new programming languages to hoops in a "quick and dirty" style, or to more easily import existing source code from other environments.

Since we envision file components being used sparingly, we do not plan more elaborate support for them.

Properties of Code Components

In addition to those properties shared by all components, code components may have the following additional properties:

Interface

The interface or signature of the component, or that part of the component that is visible to other components.

Implementation

The source code for the component's implementation, excluding its interface or signature. This applies primarily to routines.

Object code

The unlinked object code for the component. There is the need to maintain object code for different build configurations (e.g., production, debug, optimized, non-optimized, etc).

---

4. The expected capabilities of files are similar to those for MPW text files.
5. An example may be the description of "generic classes" in section 7.3.5 of *The C++ Programming Language* by Bjarne Stroustrup.

Container

The component's container, for scoping purposes. For atomic components (other than a file) the container is either a module or a class.[6]

Public/Private

Whether or not the component is private to the module or class in which it is defined. In a module, a private component is equivalent to a static function, or any other declaration that is not included in a header file. A public component is a non-static function, or other component whose interface is included in a header file. In a class, the public/private/protected attribute of a member function or data member is as defined in the C++ class.

Organizational Components

Organizational components are used to arrange other components into user-defined groups. There are two kinds of organizational components: *projects* and *libraries*. A project corresponds to all the information associated with constructing a Pink program. A program may be an application, a library, a driver, or any other executable entity recognized by the operating system. Thus the project encompasses the documentation, the source, the object code, and the resources associated with the program.

A library acts as a subproject, and is used to group together a set of classes, modules, resources, or other libraries within a project. A library can be extracted from a project, in which case the extracted library itself becomes a project. Conversely, a project can be embedded inside another project, in which case the embedded project becomes a library.

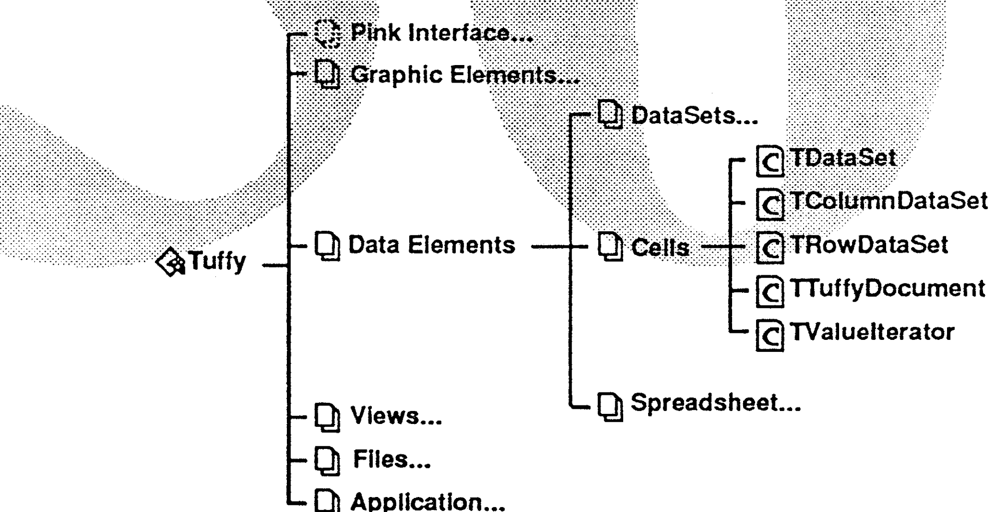


Figure 1. The structural decomposition of an application.

6. A file cannot be the container of another component because the file itself is an atomic component.

Because libraries may contain other libraries, a hierarchical structure of a program is possible. An example is shown in Figure 1 above. It shows a possible organization of SmallTuffy, the PinkMania sample application. The symbols represent libraries. In this example, the Tuffy application itself is separated into five libraries: Graphic Elements, Data Elements, Views, Files, and Application. The Pink Interface library, shown as a dimmed image, represents a library *used* by Tuffy, but not actually contained in Tuffy. The Data Elements library has been expanded to show that it contains three libraries of its own. Similarly, the Cells library has been expanded to show that it contains five classes. Tuffy itself is an application project. Hence its symbol, , is different.

Presentation of Programs

The presentation of a program is the way in which the properties of components are shown to, and manipulated by the user. hoops provides an extremely flexible way of creating a variety of presentations of a particular program or part thereof. It is based on a set of independent *viewers* that can be connected to each other in various ways.

A *viewer* displays a property of a component. An *editor* is the combination of a viewer and one or more *transformers*, which are tools that change the value of a component's property. A viewer has an *input*, which is essentially a list of (usually one) components. A viewer has an *output*, which is also a list of (usually one) components. The output of a viewer is typically whatever is selected in the view. The output of one viewer can serve as the input to another viewer. Although this is a simplified description, it follows that given a reasonable set of viewers it is possible to achieve a broad variety of useful combinations by "connecting" the output of one viewer to the input of another, and so on.

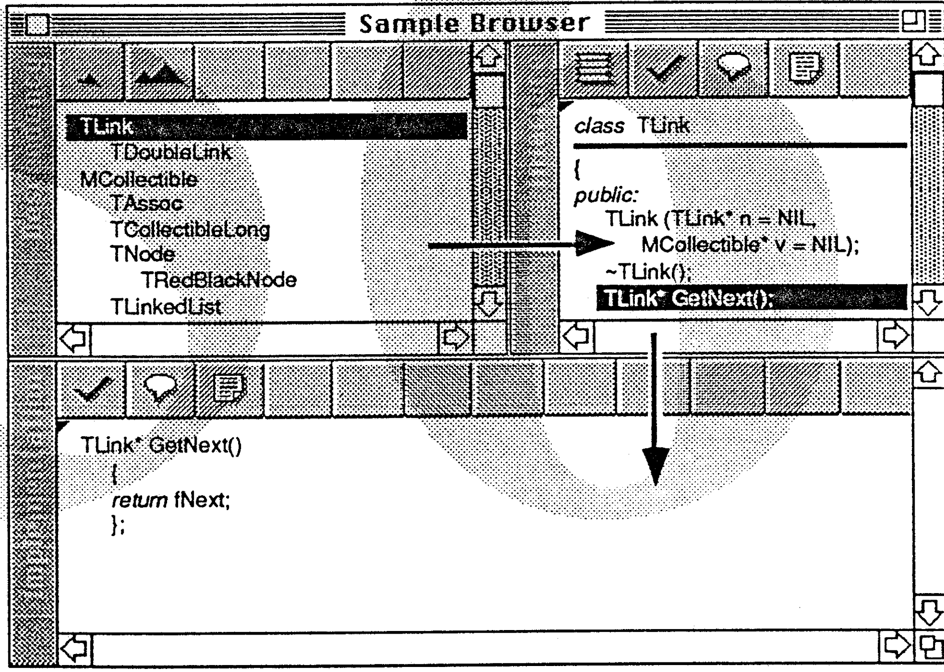
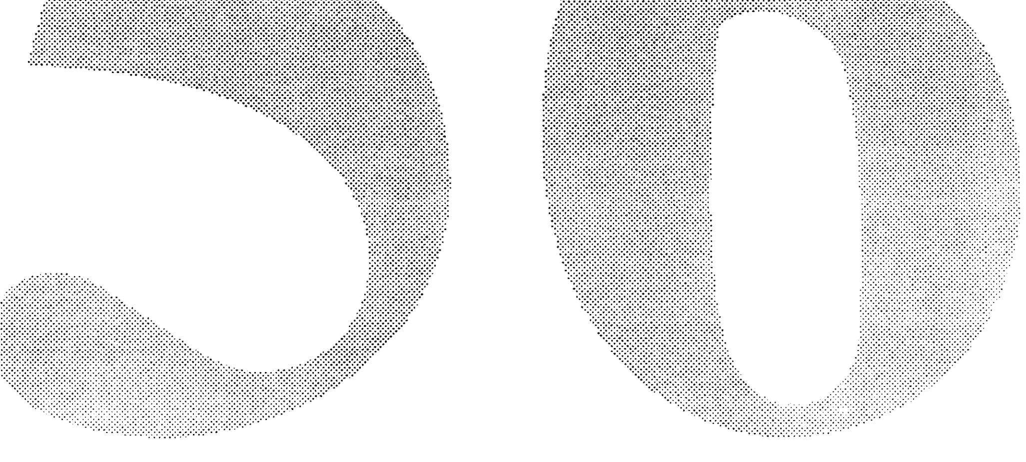


Figure 2. A sample hoops source code browser which exhibits many of the same characteristics as a Smalltalk system browser.

To show how this works, consider the browser in Figure 2 above.[7] The browser contains three *panes*, or rectangular areas that accept viewers. In the top-left pane is a class hierarchy viewer, showing a class hierarchy in outline form. Its output (the selected class) is fed to the input of the top-right pane. It contains an interface viewer that shows the interface of the class TLink. The output of the interface viewer is the selected member of the class, which is fed to the input of the implementation editor. In the example, the member function GetNext has been selected in the interface viewer: It is sent as the input to the implementation viewer, which in turn displays the implementation for GetNext.

hoops will, of course, come with a set of "assembled" browsers ready for use. However, we can neither anticipate the needs of every user, nor their personal styles. For example, some people like multipane browsers, while others prefer multiple, single pane browsers. An advantage of the scheme just described is that it's possible to provide users with an easy way to construct their own browsers, thereby tailoring hoops to different styles.[8]

Two keys to the success of this approach are providing a rich set of viewers, and an easy of way of connecting them together in new and unique combinations. "The hoops Human Interface" section of this document describes a sampling of viewers supplied with hoops, and the techniques for connecting them together.



7. This window is purely fictional and may not actually appear in hoops. Any similarity to actual windows, living or dead, is purely coincidental.
8. A further advantage over hardwiring the user interface is that it allows considerable flexibility in the design and implementation of hoops itself.

The hoops Object-Oriented Framework

We're not in Kansas anymore.
- Greyhound bus driver after crossing the state border into Oklahoma

There are two basic ways of extending hoops. The first technique may be better termed "customization" rather than extension. It involves making new and unique combinations of an existing set of raw materials (e.g., the viewers and browsers supplied with hoops). In MPW this is accomplished by writing scripts. In hoops, this is largely accomplished by creating new browsers. (See the "Dynamic Browsers" section of this document.)

The second kind of extension is accomplished by adding to, or modifying the raw materials themselves; in other words, by adding new tools or changing the behavior of existing ones. In MPW this is done by writing new tools, though it is hardly an all-encompassing solution because many parts of the environment, the MPW editor being but one example, can't be changed at all. Our approach to extending hoops is identical to that of producing Pink applications. In Pink, new applications are created by subclassing the appropriate Pink application framework classes. Likewise, in hoops new tools or environments are created by subclassing the appropriate hoops viewer framework classes.

To make this type of extension possible, hoops will provide an interface into hoops itself, which we call the *Environment Framework*. It is shipped with hoops as a library, much like the Pink interface. This library includes all of the definitions (e.g., classes, types, variables, etc.) that are available to tool builders, just like the Pink interface supplies all the definitions available to the application builder.[9] The Environment Framework includes definitions for the classes from which new tools are constructed, the classes representing program components, and an interface to the various parts of hoops which may be used by other classes. Thus, the way in which one adds new components to hoops is by subclassing the appropriate component class. Likewise, the way in which one adds new viewers or editors to hoops is by subclassing the appropriate viewer/editor class. And of course, the way in which one adds new tools to hoops is by subclassing the appropriate tool class.

Tool Classes

The tool classes can be thought of as defining both the kinds of tools that make up hoops, and specific implementations of those tools. For example, a kind of tool might be an editor, of which hoops might include specific implementations for C++ and Pascal source code. An independent tool builder could add a new editor by subclassing one of the hoops editor classes. Ideally, all of hoops is implemented in this fashion, including the compilers, linker, editors, and so on.

To allow for a broad range of extensions, the framework will provide abstractions for different kinds of tools, from which the programmer can produce specific kinds of tools. Fundamentally all tools belong to one of the following three categories:

9. For the purpose of this discussion, the term "tool" is used more broadly than in MPW, encompassing all the capabilities of hoops, including editors for example.

Viewers

Viewers represent a view of some data, as described in the "Presentation of Programs" section. Viewers are generally responsible for the visual presentation of some data, and for handling mouse clicks, selections, and other operations that change the presentation (as opposed to the data being presented). For some kinds of data, there may be several viewers, each presenting a different view of the same data.

Transformers

Transformers are tools that change a property of a component. Transformers are typically used in conjunction with viewers, though this isn't necessary—the MPW Canon tool can be thought of as a viewless transformer that transforms the source code of a component. A viewer/transformer(s) combination is called an *editor*. Another example of a transformer might be an alphabetizing tool that alphabetizes some text, where the text to be alphabetized and the resulting alphabetized text are one and the same.

Translators

Translators are tools that take component properties as input, and produce new properties of either the same component or other components. Compilers are examples of translators, in that they take the source code of a component and produce object code.

Adding New Tool Classes to hoops

While the Environment Framework itself provides a way to implement extensions to hoops, it does not address the problem of making those extensions available to hoops. To do this with today's MPW technology, one would have to relink hoops in order for it to gain access to the new classes. This is not a desirable solution. Instead, we would prefer that these new classes be made available to hoops at run-time, without relinking or otherwise changing hoops itself. Fortunately, the Pink run-time system provides exactly such a feature, called dynamic classes, which allows classes to be added to an existing application at run-time. The details of dynamic classes are still undetermined, but the basic idea is that dynamic classes are deposited in known locations to the system or an application, so that they can be found by applications that need them. Thus, when hoops starts up it would obtain access to all of the dynamic classes available to it, thereby making them part of hoops.

Given that hoops remains unchanged in this scenario, it follows that the only means of communication between hoops and the dynamic classes is via a predefined interface or protocol. This implies that the only kinds of extensions to

hoops are those permitted by the interface or protocol.[10] This is one of the reasons why it is important that the Environment Framework provide a good set of abstractions from which a broad range of tools or components can be built.

Once new tools and classes are made accessible to hoops, there still exists the problem of making them accessible to the user. A command-line interface, such as MPW's, solves this problem to a great degree because the user can simply type in the name of a tool to invoke it. Assuming that we don't want a command-line interface, we need an alternative. For viewers and editors this is done by using hoops' "dynamic browser" capabilities (see "The hoops Human Interface" section), which permit assembly of any combination of editors and viewers known to hoops. For other tools, we can associate them with the kind of data on which they operate. For example, the Count tool operates on data that can be represented as a text stream. The advantage of this approach is that tools can be made available in a context-sensitive manner. That is, when the selected data can be represented as a text stream, then the Count tool is made available in selection-specific menus.

10. Upon initial examination, this may appear to be a difficult if not impossible task. However, similar mechanisms already exist on the Macintosh. For example, all windows and controls are implemented via "definition procedures," permitting the addition of new windows and controls without changing the Window or Control Managers. Similarly, ResEdit has the "picker" mechanism for adding new resource editors to ResEdit without changing ResEdit itself. Both the Toolbox and ResEdit have invented a dynamic linking mechanism to suit their specific needs. Dynamic classes, on the other hand, are a system-wide solution available to all Pink applications.

The hoops Human Interface

No more command-line interpreters. Ever.

This chapter describes our current thoughts on the hoops human interface. Clearly this section of the ERS is subject to great change. In addition to reading this section of the ERS, you may wish to view the hoops prototype. You should also bear in mind that because hoops is a Pink application, its human interface is directly affected by the Pink human interface currently under development.

Basic Principles

Being a Pink application, hoops adheres to the general Pink human interface principles of direct manipulation, see-and-point (rather than remember-and-type), consistency, and so on. There are some hoops-specific applications of these principles:

- hoops will make use of color as deemed appropriate. Colors will be mapped to gray scales for monochrome monitors, but multibit depth pixels is *required*.
- Displayed components can be acted upon, regardless of the context in which they are displayed. For example, anywhere you see the name of a class you can obtain its source code.
- The human interface does not have a command-line interpreter. This is in line with the Pink "see-and-point" principle.
- The human interface will not *rely* on scripting. That doesn't mean scripting won't exist. Rather, one will be able to operate hoops without writing scripts. It is intended that some form of scripting will be available as a Pink system-wide feature, and hoops will take advantage of it.
- The human interface will be extensible. First, it will permit the creation of new pane arrangements within browsers, and for new connections between views in the same or separate browsers.[11] In this way, the user interface can be greatly customized to suit individual users. Additionally, the interface will be capable of supporting the addition of new tools (i.e., viewers) to hoops.

---

11. The term "pane" is used more liberally in this document than in the Macintosh Human Interface Guidelines.

Human Interface Elements

This section describes the fundamental elements of the hoops human interface; a number of which are shown in Figure 3. This is subject to change pending the definition of the Pink human interface.

Browsers

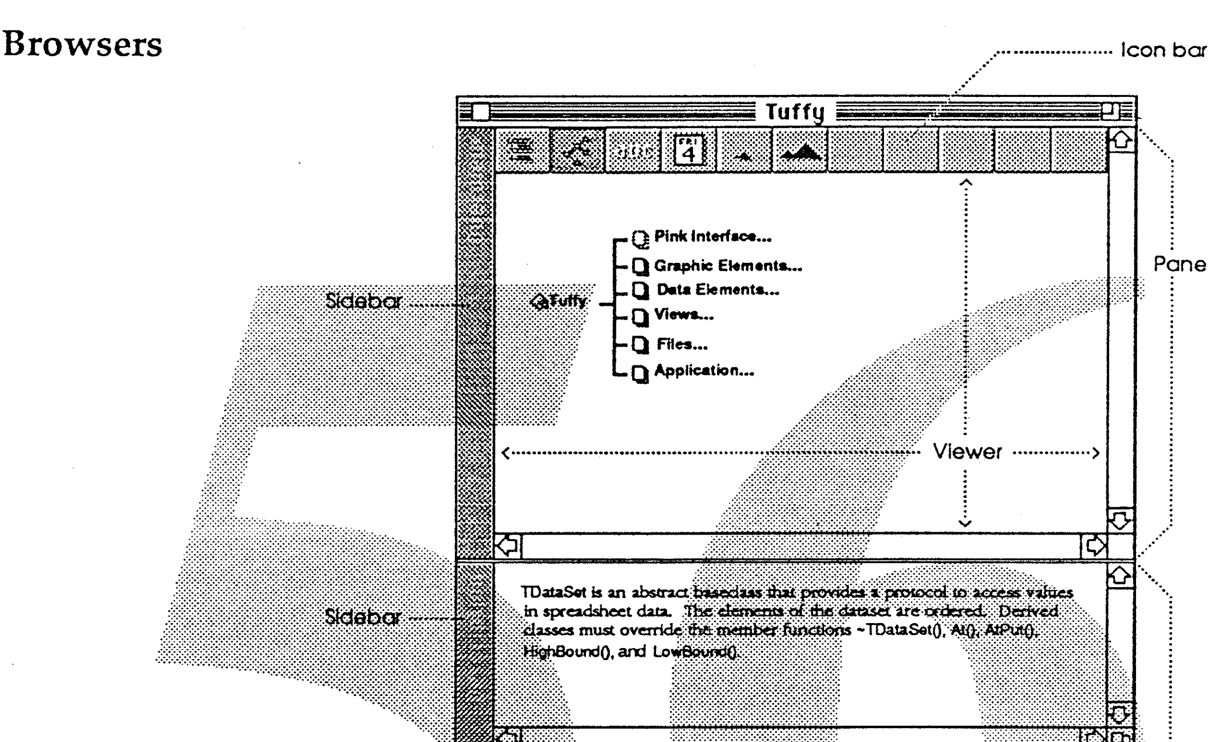


Figure 3. A description browser which includes both overview and description viewers.

Loosely speaking, a *browser* is a special hoops window that consists of one or more panes. It is often referred to as a browser because it is frequently used to "browse" a set of components.

Panes

A pane is a rectangular subdivision of a browser in which a viewer is displayed. A pane has three important parts: an optional icon bar (defined below) along the top; an optional vertical sidebar on the left side of the pane, indicating the kind of view displayed in it; and the content portion of the pane, which displays a viewer. The icon bar and vertical strip are color coded according to the view installed. The presence of horizontal and vertical scroll bars depends on the pane's viewer.

Viewers

Viewers were discussed in the Foundation section of this document. Their purpose is to display a property (or properties) of a component. Viewers are installed in panes. A pane displays one viewer at a time. A sampling of the viewers is provided below. New viewers can be added to hoops by using the Environment Framework, as described in "The Foundation."

The Main Menus

The main menus are used for commands that affect the entire environment or project, and for generic commands that apply to most selections. Examples of commands that affect the entire project might include the setting of certain options, or opening an empty browser. Cut, Copy, Paste and Print are examples of generic commands.

Selection Specific Menus

For any given selection in a view, there is a set of commands or operations that apply specifically to that selection. The set of commands or operations differs for each kind of selection, and includes those things that are sensible for the given selection. For example, the operations for a selected class might include spawning a class ancestry or class descendancy view, while the operations for a selected module would not, because the module has no relation to class hierarchies.

There are a number of possible mechanisms for implementing selection-specific commands. One way is to put *all* commands in the main menu bar and enable only those that are appropriate to the current selection. This would result in some mighty big menus. Another possibility is to add selection-specific menus at the end of the menu bar. This results in a change to the menu bar whenever the kind of selection is changed. Another solution, which we have used in our prototypes is the pop-up menu; whenever you wish to invoke a selection-specific operation on the selection, you invoke its pop-up, whose contents are always specific to the selection.

The Icon Bar

Each pane may have an icon bar along its top which includes commands that affect the pane or its view as a whole. The sorts of things one would find in the icon bar would include icons to switch views within the pane, or to invoke filters of a view, or to invoke various tools that act on a view.

A hoops Viewer Sampler

In this section we show what a number of hoops viewers might look like, and how they might operate. This set is intended to be representative rather than exhaustive. For example, hoops will also provide viewers that will display information such as "who implements method X" and "who references method X", although these viewers are not shown below.[12]

Program Organization Viewers

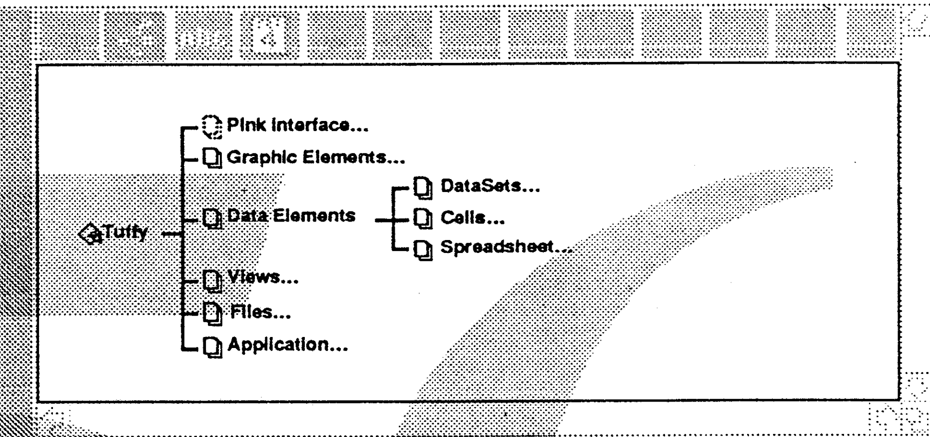


Figure 4a. A view of the Tuffy application's contents as a hierarchical structure in graphical form.

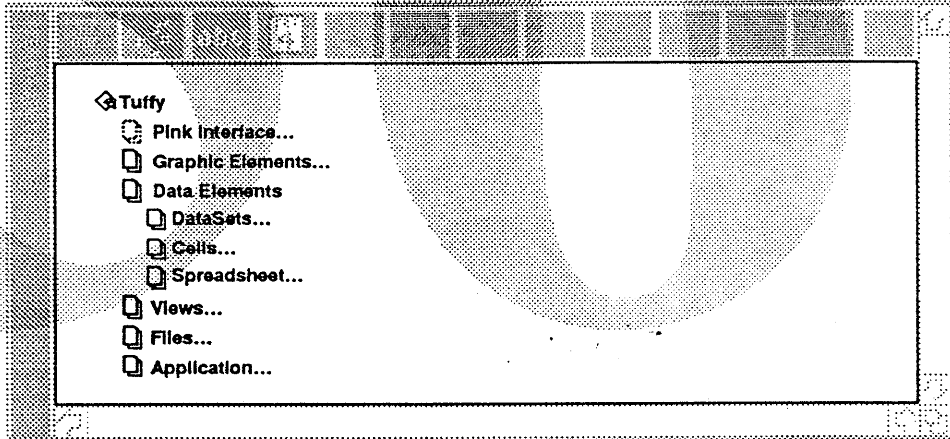


Figure 4b. A view of the Tuffy application's contents as a hierarchical structure in outline form.

---

12. These examples of views are shown with "grayed out" panes in order to both give a context for the view and to clearly differentiate the view (the dark part) from its enclosing pane. In real life, the panes and their views are equally dark.

Component	Modification Date		Modification
TDataSet	Thu, Dec 7, 1989	4:29 PM	interface, implementation
TRowDataSet	Thu, Dec 7, 1989	4:28 PM	implementation
DataSets	Thu, Dec 7, 1989	4:20 PM	interface, description
TRow	Thu, Dec 7, 1989	4:20 PM	implementation
TScatterPlot	Thu, Dec 7, 1989	3:57 PM	interface, implementation
TPlot	Wed, Dec 6, 1989	12:52 PM	interface, implementation
TColumnDataSet	Wed, Dec 6, 1989	11:48 AM	implementation
TTuffyDocument	Tue, Dec 5, 1989	10:38 AM	implementation
TValueIterator	Tue, Dec 5, 1989	10:30 AM	implementation

Figure 4c. A view of the Tuffy application's contents according to modification date.

Purpose

Displays the contents of a program, organized in a tree of libraries, modules, and classes. There are four kinds: graphical, outline, date modified, and alphabetical. The data modified and alphabetical views only show the nodes of the tree in a particular order, and are devoid of the tree hierarchy. Three of the four views are shown in Figures 4a, b, and c.

Input

Any library, or derivation thereof, including the project itself.

Output

Selected components.

Selection Techniques

Any individual node, or a set of possibly discontiguous nodes can be selected. Selecting a non-leaf node selects the node itself and the tree emanating from it. An insertion point can also be set for the insertion of a new node or the contents of the Clipboard.

Operations

Nodes can be inserted and deleted via Cut, Copy and Paste. Pasting to an insertion point inserts the Clipboard contents, while pasting to a selection performs a replacement.

In the graphical and outline views, double-clicking a node's icon expands or collapses its subtree. Node reordering within a subtree is accomplished by dragging the node's icon until it appears in the the desired location with respect to other nodes in the subtree.

A node is renamed by clicking its name and changing it directly, as one would change the name of a file in the Finder.

Given a selected node, a new viewer of the same type can be spawned in a new browser, allowing browsing of the subtree whose root is the selected node. It is possible to search for nodes of a given name.

Any browser capable of accepting the selected component(s) as its input can be opened. In that case, the selected component(s) becomes the new browser's input.

Description Viewer

The Member function determines whether its parameter is contained in the collection this. Each object in this is compared to obj using the function testFn. Member returns the object for which testFn returned true; nil is returned if no object is found.

Overridden so that if the collectible (a TCell) has a column number that matches the dataset's fColumn, and the input cell matches the one referred to by the dataset, return true. This should be more direct than scanning the whole collection, as Member may otherwise do.

Figure 5. A description viewer.

Purpose Displays and permits editing of a component's description.

Input Any single component. (All components can have a description.)

Output Selected text in the description.

Selection Standard Pink text selection techniques.

Operations Standard Pink text operations (including the incorporation of graphics).

Version History Viewer

version	author	modification date		description
6	Curt	Thu, Dec 7, 1989	4:15 PM	added GetOutOfHere()
PinkMania	Dave	Wed, Dec 6, 1989	2:32 PM	changed At(), HighBound()...
4	Dave	Wed, Dec 6, 1989	2:02 PM	changed Member()
3	Dave	Mon, Dec 4, 1989	3:13 PM	added Member()
2	Dave	Fri, Dec 1, 1989	9:38 AM	changed ~TRowDataSet
1	Dave	Tue, Nov 28, 1989	1:32 PM	new class

Figure 6. A modification date viewer displaying information about a single component's versions.

Purpose

To display the version history of a component, including the date of each version, who made the version, and a brief description of the change made in the version.

Input

Any single component.

Output

Selected version(s) of the input component.

The Display

The version column indicates the version number of each version, or a named project configuration for which the version is associated. If a component does not change between named configurations, it may be associated with more than one configuration. In that case, the first configuration name is shown, followed by "...".

The author is the individual that made the change resulting in a new version. The date is the date of the change.

The description column provides a brief indication of the type of change. hoops will do its best to automatically provide a meaningful description. For example, figure 6 shows the version history for a class. The descriptions were automatically generated by hoops, informing the user as to what member of the class was added or changed, without requiring the user to type the description.

Selection Techniques

Selecting anywhere in the line selects that version. Multiple versions can be selected by dragging through multiple lines, or by extending the existing selection.

Operations

Any browser capable of accepting the selected component(s) as its input can be opened, in which case the selected component(s) becomes the new browser's input.

Class Ancestry Viewer

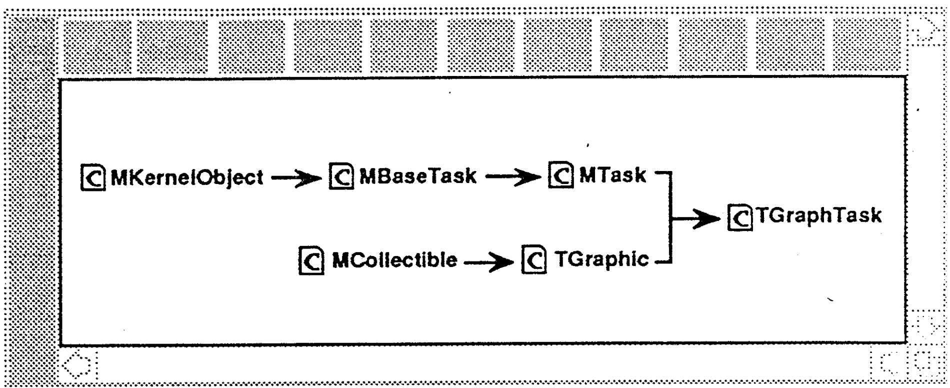


Figure 7. A class ancestry viewer displaying a view of TGraphTask's ancestors.

Purpose

To display the class ancestry of a class. The ancestry of a class is the set of classes from which the class is derived. For each class starting with the input class, the display shows all the classes included in its definition. Thus, a class may appear more than once in the display, depending on its derivation.

Input

Any single class component.

Output

Selected class(es).

Selection Techniques

Any single class, or set of classes shown in the view can be selected.

Operations

Given a selected class or classes, any browser capable of accepting a class or classes as input can be opened. Other operations are being considered such as: modifying the class hierarchy, deriving new classes, etc.

Class Descendency Viewer

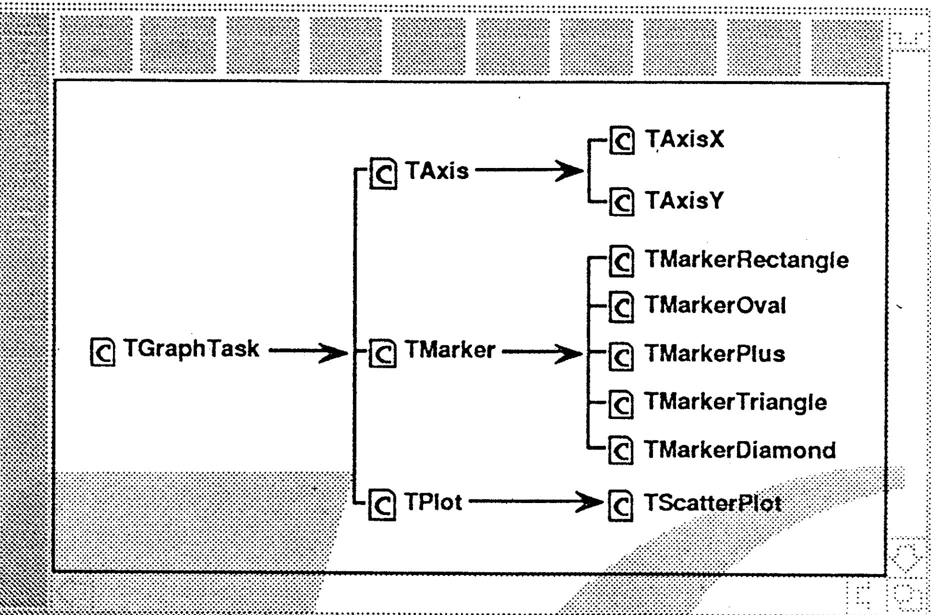


Figure 8. A class descendency viewer displaying a view of TGraphTask's descendents .

Purpose

To display as a tree the class hierarchy that descends from a given class. Portions of the hierarchy can be collapsed or expanded like the graphical program organization view. For each class starting with the input class, the display shows all the subclasses which name the superclass in its definition. Thus, a class may appear more than once in the display, depending on its derivation. (The example shows the classes descended from TGraphTask, none of which are defined with multiple inheritance.)

Input

Any single class component.

Output

Selected class(es).

Selection Techniques

Any single class, or set of classes shown in the view can be selected.

Operations

Given a selected class or classes, any browser or tool capable of accepting a class or classes as input can be opened. Other operations are being considered such as: modifying the class hierarchy, deriving new classes, etc..

Interface Viewer

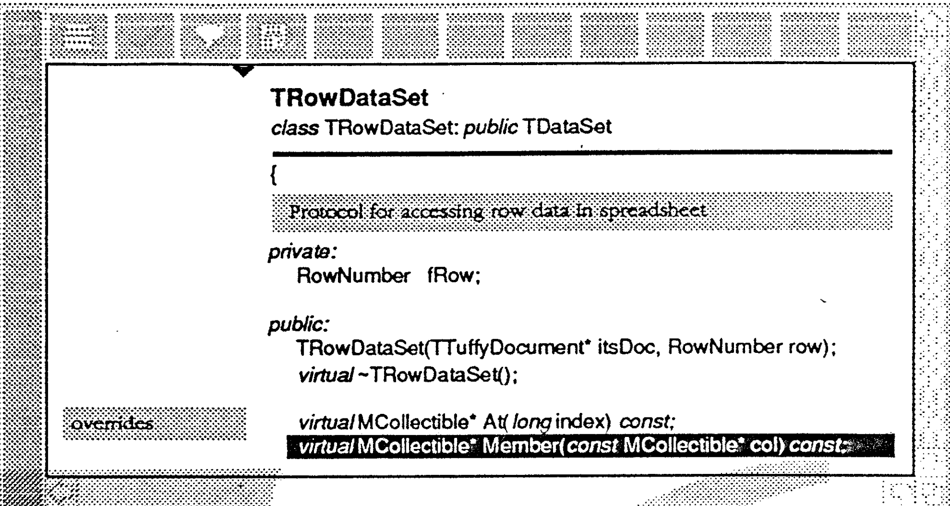


Figure 9a. An interface viewer displaying the definition of the TRowDataSet class.

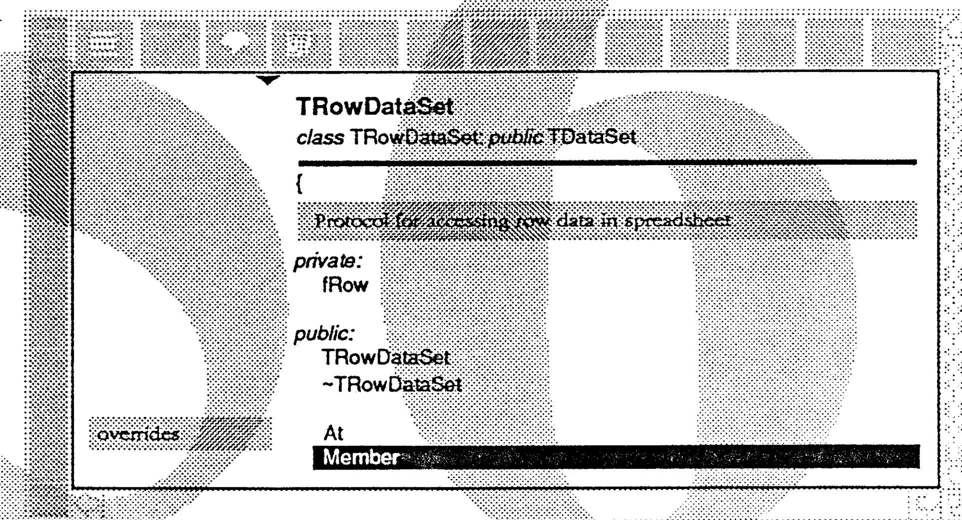


Figure 9b. An interface viewer displaying an abstract definition of the TRowDataSet class.

Purpose To display and permit editing of the interface of a component.

Input Any module or class component.

Output Any selected component(s) in the interface.

Selection Techniques Any single component, or set of components can be selected from the class or module.

Operations Given a selected component or components, any browser or tool capable of accepting the component(s) as input can be opened.

Implementation Viewer

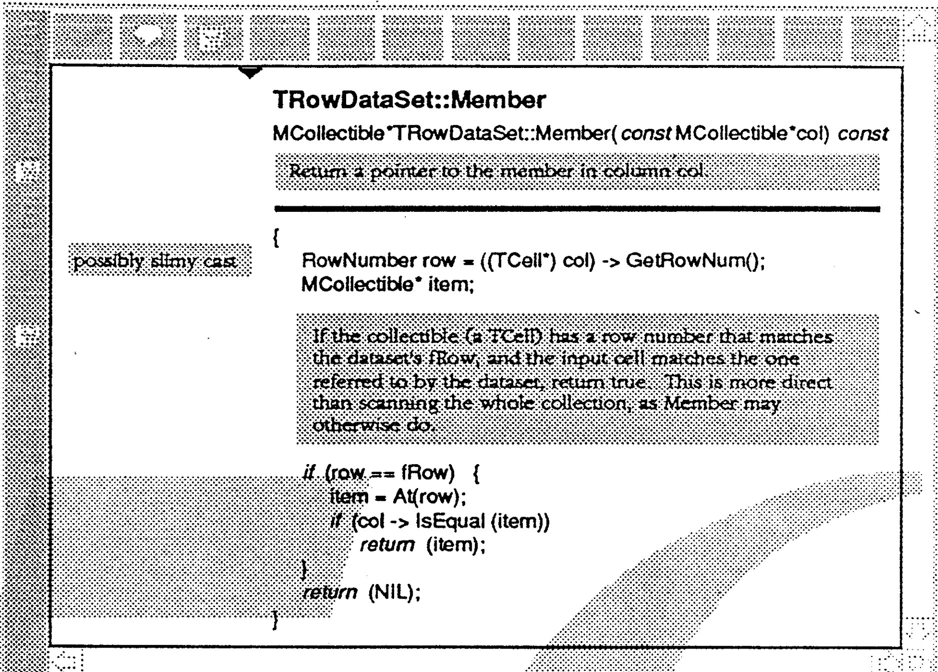


Figure 10. An implementation viewer displaying the source code for the TRowDataSetclass::Member function.

Purpose To display and permit editing of the source code of a component.

Input Any single component.

Output Selected text.

Operations See the subsection entitled "Source Code Editing" for operational details.

Object Code Viewer

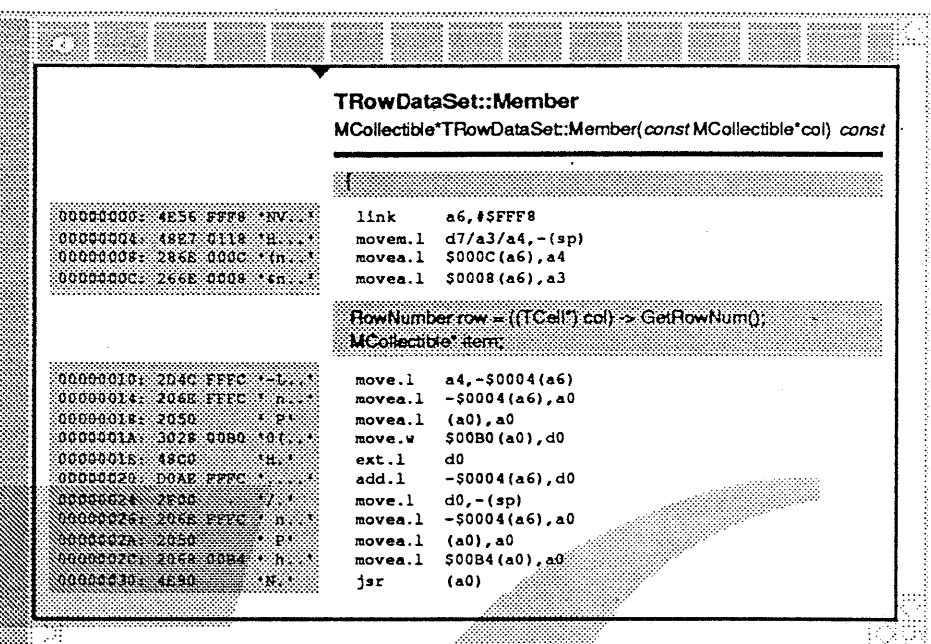


Figure 11. An object code viewer displaying the disassembled object code for the TRowDataSet::Member function.

Purpose To display the object code generated for a given component.

Input Any single component. For collection components, the object for all of its contents is shown.

Output Selected object code, represented as text.

Selection Techniques Undetermined at this time, though selection will probably be used for debugging purposes.

Operations Debugging operations will probably be associated with this viewer.

Dynamic Browsers

While hoops provides a set of "assembled" browsers, a powerful feature of hoops is that the user can change the layout of any browser. Furthermore, by saving those changes the user can modify existing kinds of browsers, or add entirely new kinds of browsers to the interface. This gives the interface a tremendous amount of flexibility and extensibility.

The feature of hoops that allows such flexibility is the treatment of viewers and panes as independent entities that can be connected to each other. In a nutshell, browsers consist of one or more rectangular areas called panes, each of which displays a viewer, and are connected so as to "drive" each other. The "layout" of a browser describes the number and location of the panes, their associated viewers, and the connections between the panes. By changing browser layouts, the user is able to create entirely new kinds of browsers beyond those supplied with hoops itself.

In order to achieve the flexibility desired, the hoops human interface addresses four areas: pane layout within browsers, pane/viewer association, pane connections, and saving browser layouts.

Pane Layout

The user can change the pane layout of a browser at any time. This is done by either changing the size of existing panes by moving their edges, or by splitting existing panes horizontally or vertically to form new panes. By default, newly formed panes take on the same properties (i.e., viewers and connections) as the pane that was split, allowing for "split views." However, since the user can change the viewers and connections of a pane, splitting a pane may also be used to add a new pane of entirely different behavior.

Viewer/Pane Association

One or more viewers may be associated with a pane. Since panes display only one view at a time, the purpose of associating multiple viewers with a single pane is to enable the user to easily "toggle" among viewers of the same data, say by choosing the appropriate icon in the icon bar. At any time the user may change the set of viewers associated with a pane by choosing from a menu or palette of viewers. This is useful for newly created panes.

Connections Between Panes

Like viewers, panes have inputs and outputs. The input of a pane becomes the input to its viewer. Likewise, the output of its viewer becomes the output of the pane. This extra level of indirection allows the pane's viewer to change without breaking the connections established among panes.

The output of a pane can be connected to the input of another, thereby connecting panes together. This is done with a wiring tool similar to Constructor's. These connections are "hot", causing the panes to "drive" each other dynamically. When the output of one pane changes, it changes the input of the panes connected to it, thereby changing the data displayed in those panes. For example, consider a two-pane browser which connects a pane containing an interface viewer and a pane containing an implementation editor. By changing the selected component in the interface viewer, its implementation is immediately displayed in the implementation editor.

The output of one pane can be connected to the inputs of several panes, in which case the same input "drives" several panes. This makes it possible to produce different displays of the same input, all of which change when the input changes. The output of several panes can be connected to the input of a single pane. This allows for viewers that combine the properties of multiple components (say, for a comparison tool).

In all browsers the input to the top-left pane does not come from one of the other panes in the browser. Rather, its input is fixed as a result of a selection in another browser via a selection-specific menu, or perhaps as a consequence of executing a command. However it should be noted that this connection is not "hot", unlike the inter-pane connections. This type of connection is demonstrated in the "User Scenario" section of this document, where a selection in one browser is used as the starting point for displaying data in a new browser.

Saving Browser Layouts

The layout of a browser (i.e., its set of panes, connections, and viewers) may be saved at any time. This allows the user to make permanent changes in the layout of hoops-supplied browsers, or to save entirely new arrangements of panes and viewers. For the purposes of defining a new browser layout, the user opens an "empty" browser which has a single pane with no viewers or connections associated with it. By using the techniques described above, the user can then create the desired pane arrangement, associate viewers with those panes, and then connect the panes. Once completed, the user has created a new kind of browser that can be saved as part of the hoops human interface.

Source Code Editing

The hoops source code browser (which consists of an interface editor and implementation editor) differs dramatically from the text editors found in most development environments, including MPW. A goal of the hoops editors (and of all views in hoops) is to produce more readable and expressive representations of programs.

In hoops, the text processing is much more like text processing in a word processor. This means there will be little or no disruption in moving documentation between the hoops environment and a more sophisticated page layout program, for example. Some layout information may change but the basic stylistic and graphic content will be unchanged.

You can use multiple fonts and styles. Comments are intelligently handled by the hoops environment, rather than having to be parsed and filtered out by the compiler. It is possible to include graphics in comments, and to include hypertext links to different parts of a project.

Syntax and Semantic Awareness

A feature of hoops' interface and implementation editors that differentiates them from text editors is their knowledge of the syntax and semantics of the code being edited. Armed with this knowledge, the editors can do a much better job of distinguishing between the various syntactic and semantic elements of source code.

Being syntactically and semantically aware requires that the editors have some knowledge of the programming language it is editing, and that the source code be parsed into syntactic elements. When new text is entered into an editor, it remains as plain text until the compiler has processed the source at build time (or when asked to reparse a particular component). If the source has compiled correctly, the editor can then reformat it according to its knowledge of the syntax and semantics of the code.

Basic Editing Layout

The basic design for text layout resembles that described in *Human Factors and Typography for More Readable Programs* by Ronald M. Baecker and Aaron Marcus. The Baecker design is not intended to be interactive, so our version differs in some details.

The implementation editor is divided into different regions, as shown in Figure 12. Across the top of the pane is the interface region. It remains anchored and is not affected by scrolling. The interface contains the name of the component being edited in large letters, the component's interface, and the component's description. The area below the interface region is vertically divided into two regions. On the left is the margin and on the right is the main body. The dividing line between these two regions can be moved by the user. The margin only contains comments. The margin scrolls with the main body, and marginal comments act as though attached to the corresponding text in the main body (this is described more fully below). The main body may contain both comments and code.

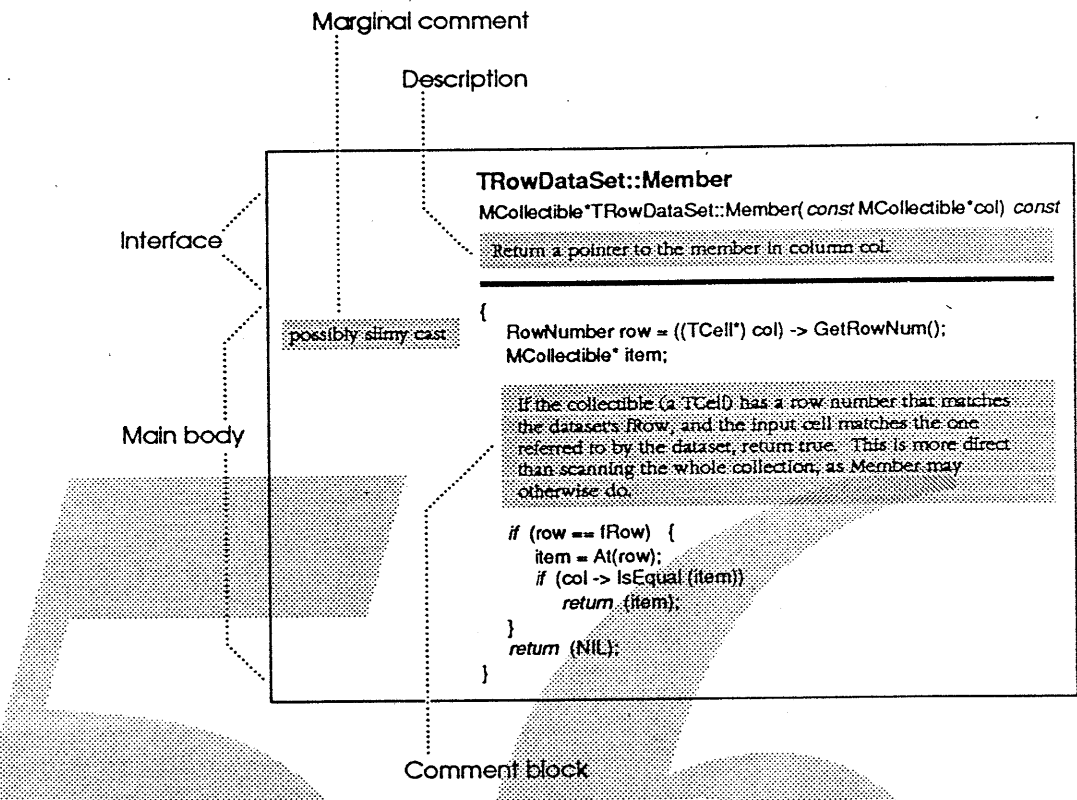


Figure 12. An implementation viewer displaying the source code for the TRowDataSet::Member function. Comments and descriptions are shown with a gray background.

Source Code Style Sheets

The hoops editor associates visual styles with different syntactic elements of a programming language. For example, in Figure 12 comments are distinguished from source code by using different fonts, and by placing the comments in a gray background. Finer visual distinction is also possible; keywords could be displayed in italic and identifiers in bold. The effect then, is to intelligently apply typesetting techniques to the presentation of source code, much like word processors do for prose.

It is important to point out that the visual style of viewed source code is independent of the source code itself, and in fact is a function of viewing the code. Therefore, it is possible for different hoops users to view the same code, but with different styles.

The set of visual styles associated with a programming language is defined in a *style sheet*. The user can define his own unique style sheet, perhaps based on a set of hoops-supplied style sheets. As mentioned, newly entered text remains as plain text and is not formatted according to the style sheet until after it has been successfully compiled.

Comments

hoops provides for three flavors of language-independent comments.[13] These are private comment blocks, marginal comments and descriptions. (language-dependent comments can of course be used, but do not share the advantages of the language-independent comments.)

Private comment blocks appear interspersed with blocks of source code, as shown in Figure 12. They are considered to be a single, connected set of word-wrapped lines, and may include pictures or other information, as permitted by the Pink text system. Private comments are considered to be part of a component's implementation, and thus are not accessible to a user unless he has access to the component's implementation (i.e., source code).

Marginal comments are confined to the margin region of the view. Each marginal comment is also a set of connected, word-wrapped lines, associated with a line of source code. Marginal comments are also considered to be part of a component's implementation.

Descriptions are a form of public comment block. They behave similarly to the private comment blocks, except they are saved in the "description" property of a component. Thus, they are not hidden in a component's implementation, or even its interface. The distinction between the description and other comments allows control over what gets published by hoops with a program, and also allows description browsers. This allows descriptions to be used as a form of on-line documentation, even for components where the user doesn't have access to source code. Furthermore, any component in hoops can have a description associated with it. For higher level components such as libraries or even classes, the descriptions might correspond to the earlier sections of an *Inside Macintosh* chapter. For data and functions, the descriptions might correspond to the 'routine descriptions' at the end of an *Inside Macintosh* chapter.

Text And Selection Dynamics

In the main body of the source (excluding the margin) both code blocks and comment blocks behave very similarly. Each word wraps upon reaching a user defined margin[14] and creates a new *block* after each carriage return.[15] Each block in the main body of the source may have a single marginal comment attached to it. The starting position of the next block is determined by whichever of the main body and the marginal comment occupies the greatest number of screen lines (remember both can independently word wrap).

Comments, both the block and marginal varieties, behave as subtexts of the code. Marginal comments are treated for the purpose of selection as being attached to the beginning of their associated main body. Thus a selection started

13. These are permanent comments. Temporarily commenting a section of code is independently accomplished using the conditional mechanism provided by hoops. By separating the permanent and temporary commenting facilities, it is possible to comment out code while still retaining its code character and associated layout.
14. We will use a fairly simple line splitting algorithm for code similar to ones used in many pretty printers, and also allow for soft returns for fine tuning. Baecker shows what can be done with even a very simple algorithm.
15. A block usually corresponds to a statement for code, and a paragraph for comments. A block only contains one carriage return at the end, even when wrapped, and might have been called a 'line'.

in code and carried across a comment block will select the entire comment block, and any marginal comments that are attached to selected blocks. A selection started inside a comment block and extended past the end of the comment will select the whole of the comment.

Typing in the main body of the source behaves dynamically in a familiar way with text wrapping at margins and carriage returns starting new blocks.[16] Block commenting is switched on and off by the same command key (which also inserts a carriage return, so that code and comment blocks will always consist of whole blocks). Marginal comments may be entered and exited using the mouse, or by using the arrow keys (remembering that the margins are treated as text at the beginning of a block).

Cut and Paste

The treatment of comments as subtexts means that a selection may consist of purely comment, or of code with embedded comments, but not of comment with embedded code.[17] This means that pasting a comment into code is straightforward, since this is just the same as if the selected text had been typed.[18] Pasting code into a comment raises some questions however. Certainly the code will be converted into a text stream and inserted as text, and any distinction between comment blocks and code blocks will be lost, but what happens if there are marginal comments? Since we are not supporting a model with nested comments we can either just drop the marginal comments or drop any layout information and incorporate them as extra text in the stream. The first alternative seems the cleanest and most useful at the moment since we can't think of why you would want the second possibility.[19]

It may also be that pasting of code into code always results in the pasted code being converted to unparsed text since the meaning of symbols may be context-sensitive. Thus pasting can be thought of as always mimicking typed input.

Printing

What about printing your program? What will it look like? The answer is undoubtedly "beautiful". However you should remember that the main aim of the hoops environment is to provide a powerful, dynamic and electronic representation of a program, along with precise and rapid navigational tools. Some of the things you will be able to do with your source in hoops have no good printed representation. For example hypertext links do not make much sense on paper.

---

16. Think of paragraphs in a word processor.
17. A comment could contain text which looked like code but it would not have been parsed by the compiler.
18. Text in the code does not get tokenized until the compiler has had a go at it.
19. And in any case if the marginal comment was required it could be separately copied.

Navigation

Getting From Place To Place

In hoops, your program consists of a collection of various components which are themselves related in various ways. In the general case a component has a source representation that is usually text in some language. There are two types of navigation to consider: local and global (or inter-component versus intra-component). In a conventional file-based system local navigation corresponds roughly to navigating within a file, whereas global navigation corresponds to locating and opening files in the Finder or with standard file.

Local navigation of a piece of flat text is done in the same way as today, using scrolling and searching. Of course in hoops even flat text need not be really flat. For example there may be hypertext links in the documentation, and possibly outliner-like text "folding". Remember though that the need to scroll and search will be greatly reduced because less material is being shown in a browser. What is often accomplished by scrolling and searching today, will be accomplished by selecting items in a tree view or list.

Global navigation means finding a particular component or moving from one component to another. Any reference to an existing component in source code can be used to bring up an appropriate browser on that component. If there are multiple such browsers possible, you may choose among them. Since collection components such as classes really consist of a list of other components, it is also possible to see them as way stations on the road to their contained components. The way in which this occurs depends of course on the particular browser being used. For example the standard source code browser supplied with hoops has two panes, one above the other (See Color Figure 1 at the end of this document). An interface viewer for the selected class is shown in the uppermost pane. The implementation of a selected member function is displayed in the lower pane. In this editor the class remains constant but the displayed functions change depending on which one is chosen in the upper pane.

References

The implementation of the function displayed in the lower pane of the class editor may be used as a starting point for navigation and source understanding. Any reference to another component (such as a call to a function) in the source may be used to bring up a browser on that component. This gives a direct manipulation use-to-definition navigation capability.

In addition to the use-to-definition capability, other navigation tools may be invoked on the selected component. These tools will build a component list that will be shown in a browser much like the class editor, except that the components listed in the top pane need not belong to the same class. The currently planned tools are "References to" and "Implementations of". For example, if a member function is selected, the user may ask to see a list of the places where this member function is referenced, or a list of classes in which this member function is defined. The results will be shown in a component list browser that has two panes. The upper pane will list the answers to the query. The bottom pane will show the implementation of the selected item. For example, if the "Implementations of" tool is invoked on the selection of `this->Draw()`, all classes that implement `Draw` would be displayed in a component list browser. The items in the list can be filtered, by selecting an icon in the icon bar, to show

more or less information based on type compatibility, base classes, or derived classes.

The state of these browsers may be saved (and probably annotated) for future reference.

User Scenario

To help visualize the way in which hoops is used, we will go through a small user scenario. Suppose there has just been a major reorganization (which never happens at Apple) and you have been assigned to work on the Tuffy team (Tuffy is a charting/graphing application under development in the Pink Application Group).

You decide to browse through the program to try and understand how it all works. When you open the Tuffy project, you see a tree view of the components that make up the entire project (Figure 13). The tree view shows that Tuffy is composed of 5 libraries: Graphic Elements, Data Elements, Views, Files, and Application. The Tuffy project also references the Pink Interface library. (This project organization has been created by the Tuffy team. hoops does not impose any arbitrary organization requirements.) The libraries may be expanded to show their contents by double-clicking their icons.

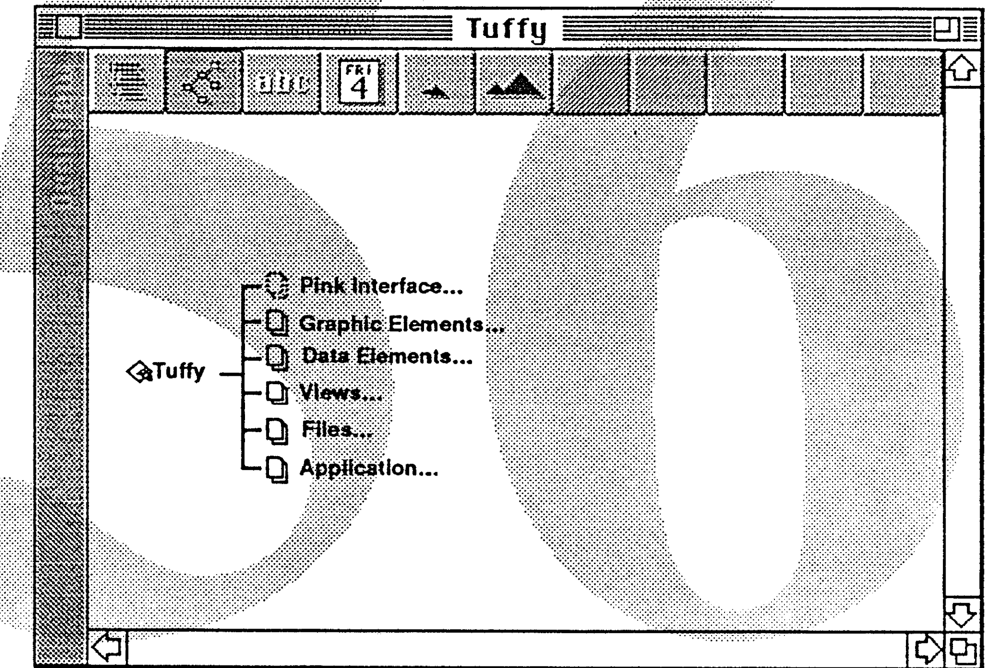


Figure 13. A first level structural overview of the Tuffy application.

Being new to the Tuffy project, you decide to explore the Data Elements Library, so you double-click its icon. Then you decide to look at the Cells library, so you double-click its icon. Clicking Cells causes the tree view to expand and show the contents. The Cells library contains 5 component classes, as shown in Figure 14. There are a number of questions you might have about the classes in the Cells library. For example, you might want to look at class definitions, examine class hierarchies, or see how the classes are used in the project. These (and other) operations are available from a context-sensitive menu. Whenever you make a selection in hoops, you may invoke a menu that will contain operations that are applicable to that selection.

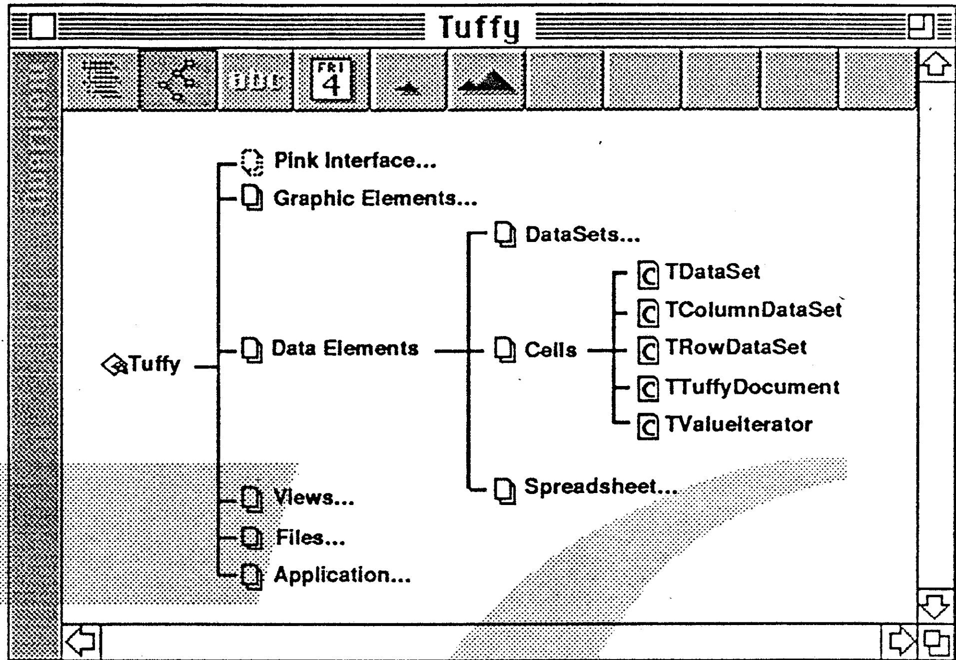


Figure 14. A structural overview of the Tuffy application. The hierarchy has been expanded to show the classes contained within the Cells library.

For example, if you selected a class, the menu might contain items such as Source, Hierarchy, and Data Connectivity. Right now, you want to see the source, so you could either select Source in the menu, or double-click on the class. Since your mind is in binary search browse mode, you select the middle class, TRowDataSet, and double-click. This opens the source code browser (see Color Figure 1 at the end of this document). The source code browser is comprised of a top and bottom pane. The top pane shows the interface of the TRowDataSet class, and the bottom pane displays the implementation of selected member functions. The interface may be displayed in different formats to help you understand the class. The default format is the plain C++ source. You select the member function Member, and its implementation is shown in the bottom pane of the viewer.

At this point, since you completely understand the TRowDataSet class, you could make a change to Member, and then accept the change and run the program by clicking the run icon. This would rebuild all components in need of rebuilding. (If you changed only the body of Member, and not its interface, then only Member would need rebuilding.) Or alternatively, you could continue to browse using Member as the starting point. Looking at the implementation, you see that IsEqual is used, and you would like to know what other functions call IsEqual. You select IsEqual, and invoke the selection-specific menu[20], which now lists items such as "Implementations of" and "References to". Selecting "References to" brings up a browser that lists all functions that call IsEqual. Using this new browser, you can navigate to other classes and member

20. If you're asking yourself, "what is a selection-specific menu?", look back on page 22.

functions. In this manner, you may explore interactions between the components in the entire Project.

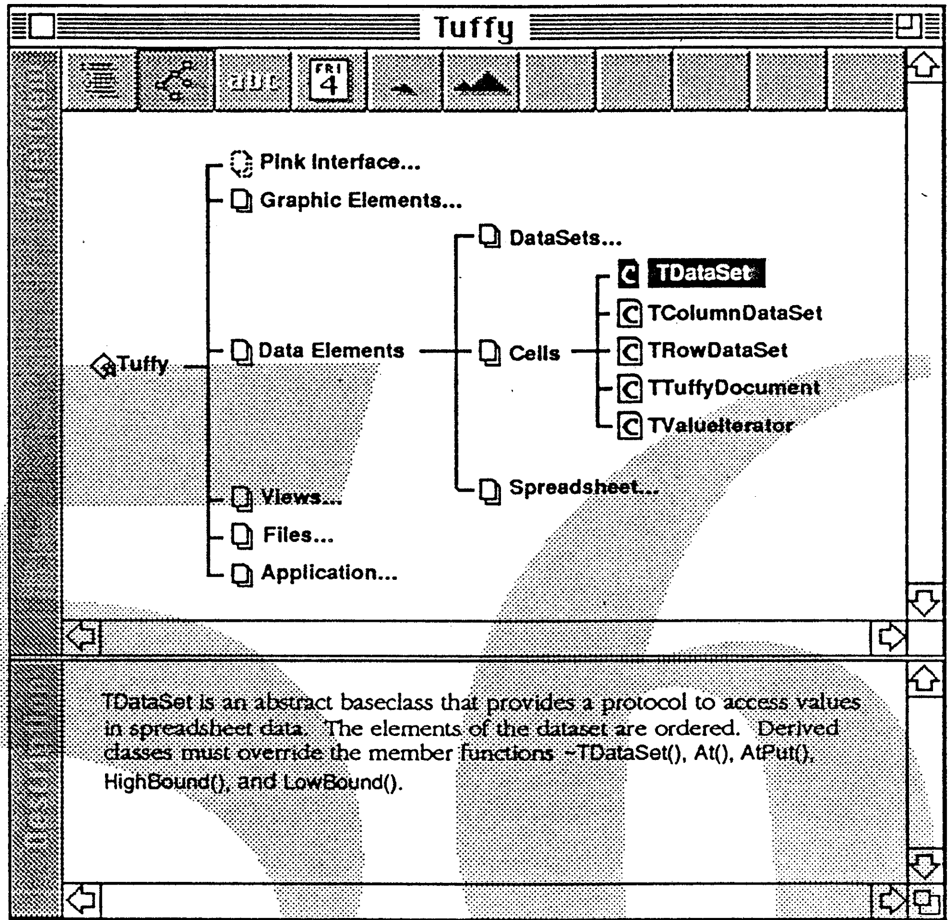


Figure 15. A description browser for the Tuffy application.

After experimenting with Tuffy, you find that you really didn't understand the class TRowDataSet after all. What you want to do now is browse through all the component descriptions in the Tuffy project without opening interface or implementation viewers on each component. Suppose that hoops does not provide such a browser. You will need to construct the browser yourself. To do this, you could open an "empty" browser, split it into two panes, install an organization viewer and a description viewer, and then connect them together. You now have a new browser that displays the descriptions corresponding to the selection in the organization tree. Using this browser is no different than using browsers that come with hoops, and in fact could be saved as part of your hoops user interface. Figure 15 shows how your constructed browser might look after selecting the TDataSet class.

Program Management

Projects

Your Program is Your Project

In hoops, Program Management is meant to cover all aspects of data associated with a programming project, including documentation, source, object code and dependency information, and how this information is used to control the building of actual programs. Thus program management in hoops covers topics that were handled separately by Projector, make and the C preprocessor in MPW.

Projects

A hoops *project* corresponds to *all* the information associated with constructing a Pink program. A program may be an application or a library or any other executable entity recognized by the operating system. Thus the project encompasses, in one operating system file, the documentation, the source, the object code and the resources associated with the program.

Multiple projects may be opened at the same time, and you will be able to make use of standard editing operations to move information between projects.[21]

Referenced Projects

Projects may refer to other projects. For example all Pink applications will refer to the Pink project in the form of the shared Pink libraries. Such a reference will not be able to access all the information used in the creation of the Pink libraries, only the published interfaces and documentation. From the programmer's point of view, referenced projects will appear to be fully integrated with his own project.

References to other projects take the place of the concept of a subproject. Projects are hoops components also, and as such, are subject to dependency control. Hence any changes in referenced projects will cause rebuilds as appropriate. A super project may be created by making references to a number of other projects in the super project. Such referenced projects may be thought of as libraries. They usually come in two flavors: *shared* and *embedded*.[22] Shared libraries are always remotely referenced and accessed and their code is not directly incorporated in the program image. Embedded libraries are pieces of code that become physically part of the program image.

References to shared libraries will cause rebuilds when the library interfaces change, but not when it's only the library's object code that changes. References to embedded libraries will also cause rebuilds when the object code of the library changes.[23]

---

21. Although this may result in some context-sensitive information (such as object code) being lost.
22. It may also be necessary to allow mixtures of shared and embedded.
23. Within a single project any references during any rebuild are to a specific version of a referenced project so rebuilds are deterministic even when circular references exist among projects.

Stationery

hoops follows the Pink model of *stationery pads* as the starting point of work. This works very well, since in the Pink environment, program projects always start with some framework which depends on the type of program[24] being constructed. For example, a Pink application always starts with the application framework. A program that conforms to the application framework but which does nothing is the *empty application* and is ultimately the starting point from which all applications come. This is what results when an empty application is "peeled off" the application stationery pad. This empty application will be fully documented as to how it is expected to be specialized, and serves one of the roles that sample applications do in today's world.[25]

The "document" created by hoops from a hoops stationery pad is always a functional but essentially "do nothing" example of the type of program represented by the pad. In this model "programs" recognized by the Pink operating system are *all* hoops documents (although published programs[26] might contain no information other than object code and resources, much as is the case in today's Macintosh).

Any project can be turned into a stationery pad, so that customized pads are easily created. Stationery pads will be provided for the most common programming tasks, such as creating applications and shared libraries. User created pads serve as a formalization of what is current common practice, namely taking an existing program and using it as the starting point of a new similar program.

24. Programs are applications, libraries or any other executable entities.
25. This does not completely do away with the idea of sample programs since these demonstrate ways of adding more functionality.
26. See the next section.

Publishing

Publishing Is Not Just Copying

While you are developing a program, hoops will be cooperating with the Pink runtime so that you don't have to undergo a full load and dynamic relink of your program[27] after each rebuild. Because of the way hoops accomplishes this, and also because of the requirements of incremental compiling and linking, the object code of your program may not be always packed or arranged most optimally. Also your program as far as hoops is concerned consists of everything, including source, that you have been using to construct it.

What this means is that hoops needs to be able to *publish* a program. When this happens hoops will rearrange and relink all the object code, and strip out any private data associated with the program. Thus hoops provides an automatic way of preparing a program for distribution.

Private data

Within a hoops project you will be able to specify whether certain data is private or public. For example in most cases the source code of your program will be private. However in a shared library you will want certain interface elements and their associated descriptions to be public, while still keeping the implementation hidden. In still other cases you may wish to make some source code available. For example, some sample code showing examples of subclassing from the Pink libraries might be shipped as part of the libraries.[28] Yet another example where data might be kept private is hoops "stationery", from which you start all new programming projects. The programs that result from the stationery are like sample programs and should support a fairly high degree of public commenting. However even here the original developer of the stationery might have private comments or implementation details.

27. With the Pink ROM libraries for example.
28. We are not addressing the specifics of whether the developer interfaces to the Pink libraries are actually part of the ROM image or are physically separate and only conceptually integrated.

Conditionals

Conditionals Are System Supported

In traditional environments, providing for conditional compilation is usually handled by written instructions to the compiler. These are in the form of some meta-compiler-language.[29] This means that conditionals can only be known to the environment in a very limited way.

hoops needs to have much more knowledge and control over conditionals in order to track dependencies and correctly automate the build process. Furthermore the traditional approach leads to a multitude of individual solutions, one per language, in a multilanguage environment.

hoops provides environmentally supported conditionals, via a direct manipulation interface. The hoops approach is uniformly applied across all languages in hoops, and participates directly in the dependency mechanism.

hoops conditionals can be used to support the creation of multiple configurations of a program from the "same" source. This allows for example, debug and nondebug versions, or versions for different processors.

Conditionals Are Just Liberated Comments

Comments are also system supported in hoops in distinction to the traditional approach. Comments are sections of source text that are never interpreted by the compiler. In hoops this means the compiler never actually sees them. It is possible to think of comments as conditional sections that are always false. In practice this is often turned around and conditionals are used to temporarily comment out sections of source.

Conditionals may be thought of as comment blocks which have flags other than "always false" associated with them. Thus conditionals always act like embedded subtexts of the source code with respect to selection. They may be expanded and collapsed just like comments, and when collapsed they are visible as simple icons, so that their presence is always apparent. Thus the human interface to conditionals is essentially the same as for comments in hoops.

Each conditional is attached to a condition or flag, which is in turn attached to a component. Conditions may be any boolean expressions, not just simple flags. Conditions are component properties and as such are known to the dependency mechanism. When a condition is true the conditional text is visible to the compiler, and when the condition is false it is hidden. The visibility for the programmer is also controllable. Each conditional can be enabled or disabled much like enabling or disabling a font style. It is possible to view the source with any number of conditions enabled, including all and none.

Conditionals can have different style attributes attached to them. For example it may be that a common condition such as "debug" might be associated with red text (or background) and that when code is being viewed with "debug" enabled, all the debugging code would appear as red.[30] Figure 16 shows some

---

29. Preprocessor instructions in the case of C and C++.
30. "I see red, I see red, I see red" - Tim Finn. Sorry, I couldn't help myself.

code with conditional debugging code (unfortunately not in red), with the flags appearing in the margins.

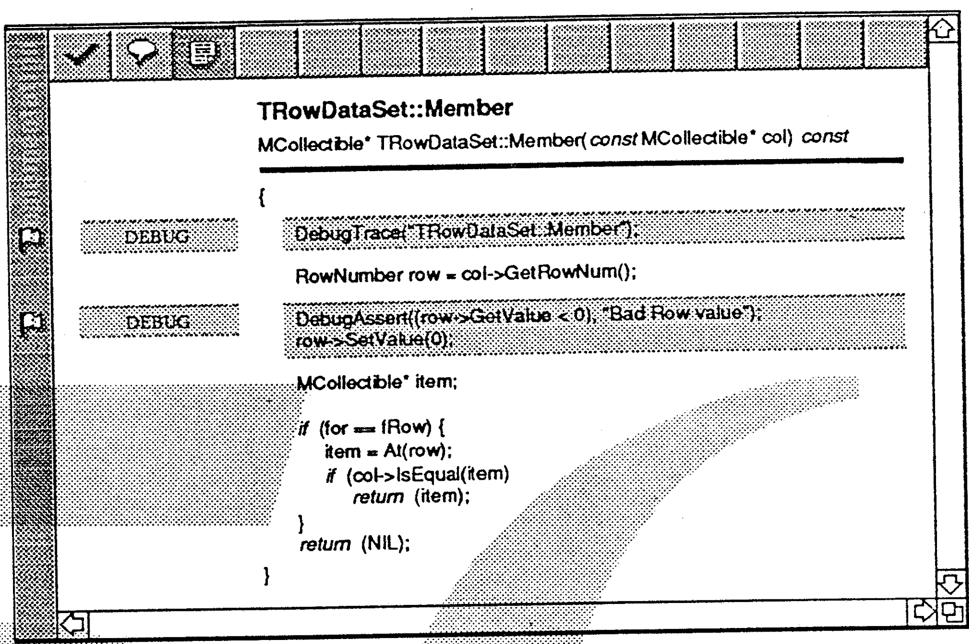


Figure 16. An implementation viewer for a member function that includes conditional debugging code.

Nested Conditionals

Conditionals may be nested, but not overlapped. This means that when two conditionals intersect, one is always either equal to or completely contained within the other. If the condition on the outer conditional is not met then none of the contents of the enclosed conditionals can be seen, regardless of the condition settings. Thus if the enclosed condition were true but not the enclosing condition, then the enclosed conditional would not be visible.

This may seem like a restriction but in fact the full range of logical arrangements is possible. Intersected conditionals represent the AND operation and non-intersected conditionals represent the OR operation.

Adding New Conditions

There will be some hoops defined conditions such as debug/nodebug and public/private. New conditions can be added by users on a component basis, to cater for special needs. When a component is created it inherits the condition state of its parent (containing) component. Conditions can be added and removed from components, and their state may be changed at any time.

Switching Conditions

You can switch the state of a condition at the component level and also globally at the program level. You can also affect a collection of components, for example, a complete library. The rule for this is that switching the state of a condition causes that condition to be set to the same state for all contained components.

Building Different Configurations

Making a change to the value of a condition on a component is considered a change by the dependency mechanism, and will trigger a recompile for that component on the next rebuild, provided its state has changed from the last rebuild. If the state is the same as at last rebuild then that component is not recompiled even if the state has been changed many times between the two rebuilds.

Configurations

Versions and Configurations

Each component may have any number of versions. A version of a component corresponds approximately to a saved version of a file in a traditional system. A big difference is that hoops keeps all old versions (at least until you decide to remove them, or run out of space). A version corresponds to the state of a component at a particular well-defined time.

The set of versions of components that go to make up the state of your project at any given time is called a *configuration*.[31] hoops records the configuration of your project (usually automatically), and this can be thought of as a snapshot of the entire project's state at a particular point in time.

Recording of configurations occurs automatically at certain natural breakpoints such as a build, or when quitting hoops. Recording of a configuration can also be forced manually at any time. Configurations are automatically date stamped, but can be renamed at any time to reflect some more meaningful checkpoint (such as "beta1" or "final").

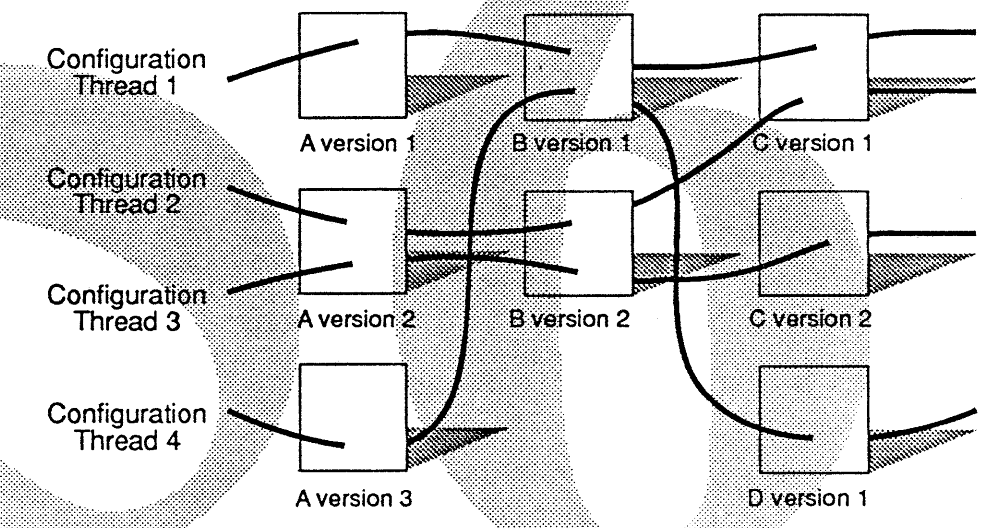


Figure 17. Configuration threads for different builds of an application.

In the model being described you never directly save or delete a version of an arbitrary component. Rather you always deal with a configuration.[32] A version of a component is never saved without saving the configuration in which

31. There is another broader usage of configuration, namely a set of rules that can be used to describe a set of configurations in the above sense. Expect terminology (and even concepts) to change in this area.
32. In addition to manipulating configurations of versions, whatever Pink provides for versions, including the human interface for manipulating them, will be what you'll get in hoops.

that version exists. Of course if you change just a single component and then save the configuration, this is much like saving a version of the component.[33]

Figure 17 shows a project that initially consists of three components (what they are doesn't matter). These are version 1 of each of components A, B and C which together form Configuration 1 of the project. A and B are both changed to version 2. Configuration 2 then consists of version 2 of A and B and version 1 of C. Configuration 4 shows how it is possible to add a new component and revert a component to an earlier version.

Configurations and Branches

You may inspect the version history of an individual component. You may even choose to revert to an older version of the component. This results in a new configuration, but of course since configurations are saved this leaves the previous configuration accessible. Since you may only work on one configuration of your project at a time, the totality of configurations is a linear list ordered by time.

Something like branches in Projector are possible merely by creating a new configuration (and renaming it). Subsequent changes are then saved in new configurations of this new "branch". Changing to another branch is accomplished by reverting to a previous configuration. Merging branches is accomplished by selecting one configuration as the parent and choosing versions of the other configuration for individual components. A standard h*oo*ps supplied dynamic browser can be used to help with the merge process.

Undo

Undo in Pink is separate from the versioning mechanism. A chain of actions may be undone back to the last configuration save checkpoint. Prior to this last checkpoint changes can only be reverted through discrete configurations, and individual actions have been lost. After each build, or other configuration save checkpoint, the undo chain is restarted.[34] There are several reasons why the undo chain is not the same thing as the versioning mechanism. The main reason is that undoing is associated with a sequence of actions which may be carried out on many components, while versions are always of particular components and may be accessed out of sequence at a later time.

33. h*oo*ps does not actually resave all the different components in your project if they haven't changed. h*oo*ps does save the changed components and the configuration information that allows the correct set of versions to be accessed when required.

34. Changed components have generated new versions which are recorded in the new configuration.

The Build Process

Automated Builds

h*oo*ps is an incremental system. The level of granularity of an increment is the component. This is somewhat similar to a traditional file-based system, where the increment size is the file. Of course h*oo*ps is also nothing like a file-based system, because all the information necessary to track and control dependencies is stored as an integral part of each component. h*oo*ps is not continuously rebuilding, but only rebuilds when asked to, exactly like a traditional system. The difference lies in the amount of work that occurs at rebuild time.

h*oo*ps will do its best to minimize the amount of reprocessing after any changes have been made in the program source or in persistent objects associated with the program. Thus changing the implementation of a function will result in an almost instantaneous rebuild, while changing the interface of a function will necessitate recompiling just the users of the function.[35] You should keep in mind however that because of the nature of static languages such as C++, sometimes apparently small changes may require a large amount of recompilation. This is because although the language supports encapsulation abstractly in the program source, it often breaks it concretely in the resultant object code. For example, adding a data member of a class will cause recompilation of any client that embeds any object from that class, even when all access to data members in the object has been made indirectly through virtual getters and setters.[36]

How?

Components are separately compilable units in h*oo*ps in the same sense that a file is separately compilable in a traditional environment. A component may make references to other components. All such external references are stored for each component as dependencies. Each component also keeps a list of its clients. This would seem to be redundant since collectively these two lists contain the same information. However since we use this information to control the build process we want to be able to follow a trail of dependencies without having to check every component in a project.

When you make a change to a component, h*oo*ps will use the stored dependencies to determine what needs to be done to accomplish a build. This will usually mean at least a recompilation of the component in question.[37] In many cases there will also be a need to recompile other components, although sometimes, for example if the change has been just to the implementation of a function, this will not be necessary. If a change has been made to an interface of a component that is used elsewhere in the project, then, unless appropriate editing changes have been made already to the component's clients, compilation will just result in compilation errors in the clients.

If all changed or dependent components compile correctly, then h*oo*ps will automatically relink any code that has changed. Once again the stored dependencies are used to minimize the amount of new work done by the linker.

---

35. Of course you will usually have edited all these users anyway to avoid generating errors.
36. And of course inline getters and setters break encapsulation all over the place.
37. Although, for example, h*oo*ps will know when a change is just to a comment and will avoid rebuilds in this case.

Error Reporting

In the best of all possible worlds you would never make mistakes, and so would not need to consider how to handle inconsistent and incomplete projects.[38] However at least some of us live in something short of utopia and so need to consider errors.

You might for example reference a variable or function that does not already exist or is not visible from your source. You will sometimes make syntax errors, and sometimes you will forget to update clients of a function whose interface you have just changed. When you try to build a program with errors, hoops will mark your source at the location of the error, so that you never have to match errors to files and line numbers. You can think of these error markers as being sticky, something like markers in MPW.[39] You will be able to access a list of all current errors and navigate directly to the location of any error. You will be able to validate errors individually so that they are removed from the list of current errors.[40] The list of current errors is completely updated after every new build.

38. It has been noted that this would not actually be the best of all possible worlds, since it implies that we have to leave our projects in a consistent state before leaving work each day.
39. The exact appearance of error markers has yet to be determined, but you will have direct access to the corresponding error message (without having to match error ID numbers to lists in some book).
40. Of course its up to you whether you actually correct them.

Importing and Exporting

Importing

hoops will provide at least a simple form of source code importation. Since hoops represents programs in a database, code produced in other environments must be imported into the database. The easiest way to import source is to convert the text files into hoops file components. Since there is an exact mapping from a text file to a file component (the only difference being that the file is an operating system object, while the component is a hoops object), it is not necessary to modify the source code in any way. Of course, file components make only limited use of many hoops features, so it would be nice to convert source code into program components of finer granularity. Unfortunately, this may not be possible for all cases without programmer modification. The problem is that it may not be possible to map all external source files into hoops program components (other than a file), because the lexical scope of files has a great effect on the semantic meaning of its contents. Examples of this problem may include order dependencies in a source file.[41] In these cases the best that can be done is to map the text file into components as closely as possible, flagging those constructs that couldn't be resolved by the importer.

Exporting

It will be possible to extract code from hoops, in the form of ASCII text files. In the case of C++ these files will be guaranteed to be syntactically correct C++, suitable for compilation by other C++ compilers.[42] Some auxiliary information, such as pictures in comments, or font styles, will be lost in the translation to ASCII text, as such information isn't representable in ASCII text.

---

41. In particular, certain uses of the C++ preprocessor.
42. Assuming they support the same version of C++ we do.

Collaboration

What Is Collaboration?

Collaboration in the hoops context means support for team programming. We want teams of programmers to be able to work on the same project with hoops keeping track of who is changing what, while making sure that changes don't get lost or out of synchronization. This certainly implies that a master version of a project is kept somewhere. A user may have the illusion that they are directly accessing the information over a network but almost certainly they will be mostly using a local copy of the information. Sometimes they will want to actually disconnect from direct connection with the master copy but still continue to work on the project remotely.

hoops intends to make heavy use of the basic facilities in Pink to accomplish its aid for team programming. This will be done in a way that should be transparent to an individual using hoops, and should not require members of a team to be in constant connection with the master project. Each of these issues is addressed individually below.

Pink Itself Is Collaborative

The Pink application engine allows multiple users connected on a network to work on a document together. hoops will follow the basic Pink model for this aspect of the document architecture.

The basic needs of collaborative projects are that different users should be able to gain write control of portions of the project, so that they can make changes. They should also be able to browse all portions of the project. Since in Pink a document does not equate to a file[43], there is no problem about tailoring the granularity of a "document" to the appropriate level required by hoops. This working granularity will be much finer than at present[44] and will be at least on the component level.

For collection components we will have to define rules for how write control is shared or inherited. For example it seems correct that gaining write control of a collection component should gain control of all its "subcomponents". However does this imply that if someone has control of a single "subcomponent" this prevents gaining write control of the parent component? And if so, is this convenient? Our choices here will probably need to be tailored to the best dynamics of use, and we expect this to be subject to tuning.

Users sharing work should be able to communicate with each other. Again this is expected to be a system-wide Pink service and hoops will use whatever system model is decided upon.

---

43. There may be many files in a document, or many documents in a file. The later case is the one most relevant to the hoops model of a programming project.
44. In Projector the level of granularity is the file, even when the unit of change is much smaller. This has the disadvantage of locking up much greater portions of a collaborative project than are necessary whenever a change is being made.

But Won't It Be Hard?

We want gaining write control to be much more direct than is possible in Projector for example. For example we want an individual user to gain the advantages of a versioning system without having to be concerned with collaboration.[45] We also want users in shared projects to work as much as possible as individuals, becoming aware of the collaborative nature of their work only when h*oo*ps cannot automatically "do the right thing" or where explicit choices need to be made. It should be possible to see the lock state of a component at all times in a joint project shared over a network, and it should be possible to just start making changes in a component if it is not locked, without having to go through a question and answer interface. h*oo*ps should take care of locking the component and transmitting the new lock state to other users. Only when a changed component is being "checked in" is there need for a user to be explicitly reminded of what is occurring and to back out of changes if desired.

I Want To Work At Home

h*oo*ps allows collaborative work at remote machines and not just while connected via the network. A user is able to "check out" a project image and work on this image completely separately from the master project.

The simplest way to accomplish this is for the user to select the components to write control and to lock these until the project is "checked in". This will probably not work well in practice however, since it requires too much fore-knowledge on the user's part. Thus we need to allow a user to take write control of a component while accessing a "checked out" project. This then makes it possible for multiple users to change the same component "simultaneously". There is clearly no automatic way of reconciling multiple changes of this sort. It is however possible to track the change state of a component and make sure that a user never *unknowingly* "checks in" a changed component over the top of someone else's changes.[46] By ensuring that users have full knowledge and by providing assistance in merging changes, h*oo*ps intends to make this situation as painless as possible. Remember too that changes are tracked by the versioning system, making it harder to completely lose the changes of an individual.

One possible problem is that even in the shared case, versioning is different from "check in". A component may go through a number of versions between being "checked out" for writing and being "checked in". Other users should not even be aware of these intermediate versions and while they should be notified of "check ins" they should probably not be *forced* to update to the latest configuration.

---

45. In Projector these two functions are mixed.
46. Of course the first user to "check in" cannot know about the other users unless they explicitly requested write control.

Debugging

Debugging in hoops is accomplished using many parts of the entire system rather than invoking a special debugger. Support for inspecting data, controlling program flow, monitoring progress, etc., is integrated into the environment to simplify the process. These functions are performed by pointing and clicking. Typing is used only for entering new data. In addition to the standard debugging features, hoops takes advantage of its knowledge of program structure to support program analysis tools not typically found in other environments.

Inspecting

Inspecting program data is a major strength of the hoops environment. The symbolic information available to the system is used to support natural displays of data. Several methods of inspection are now being considered. One prototype displays data using an inspector (Figure 18) which shows the contents of the current task's stack. The stack is shown in outline format, initially collapsed, with the outermost level corresponding to a stack frame. Frames can be expanded or contracted in place to reveal their contents. A structured variable within a stack frame can also be expanded and contracted in a similar manner. Pointers can be clicked to open other inspectors displaying the objects being pointed at.

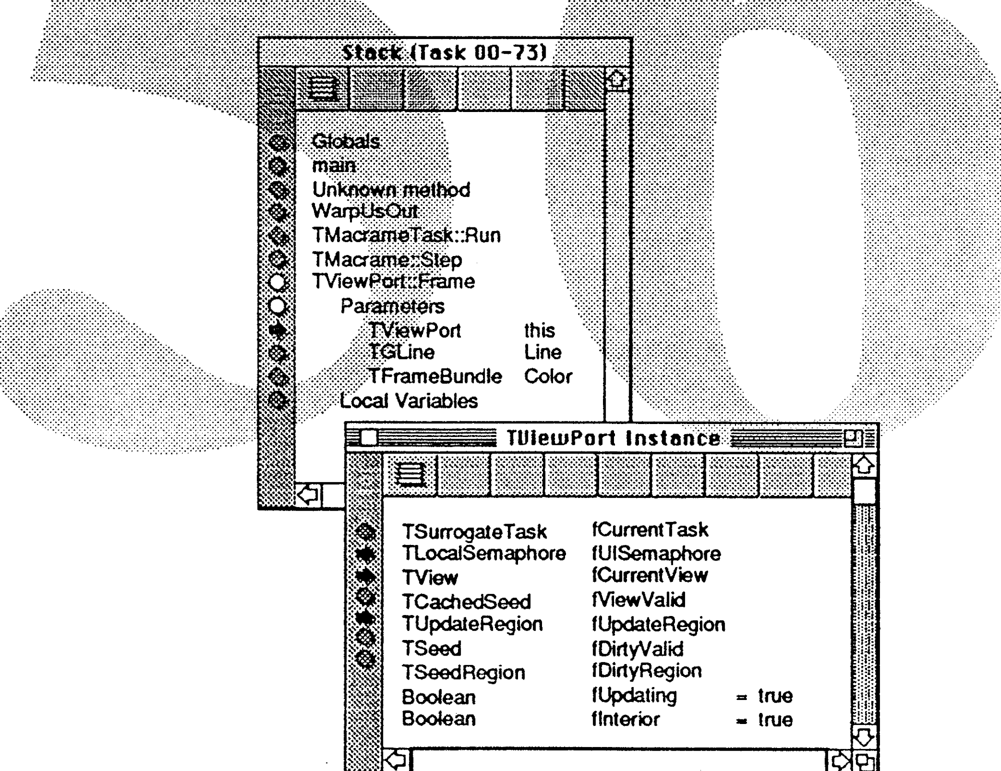


Figure 18. Two inspectors: one showing the contents of the user's stack; the other, the contents of an instance of a TViewPort which was referenced on the stack.

Another method uses more of a browser approach to data display (see Figure 19). Separate panes showing stack, stack frame, source, etc., are connected together to provide a point-and-click method of displaying the appropriate variables.

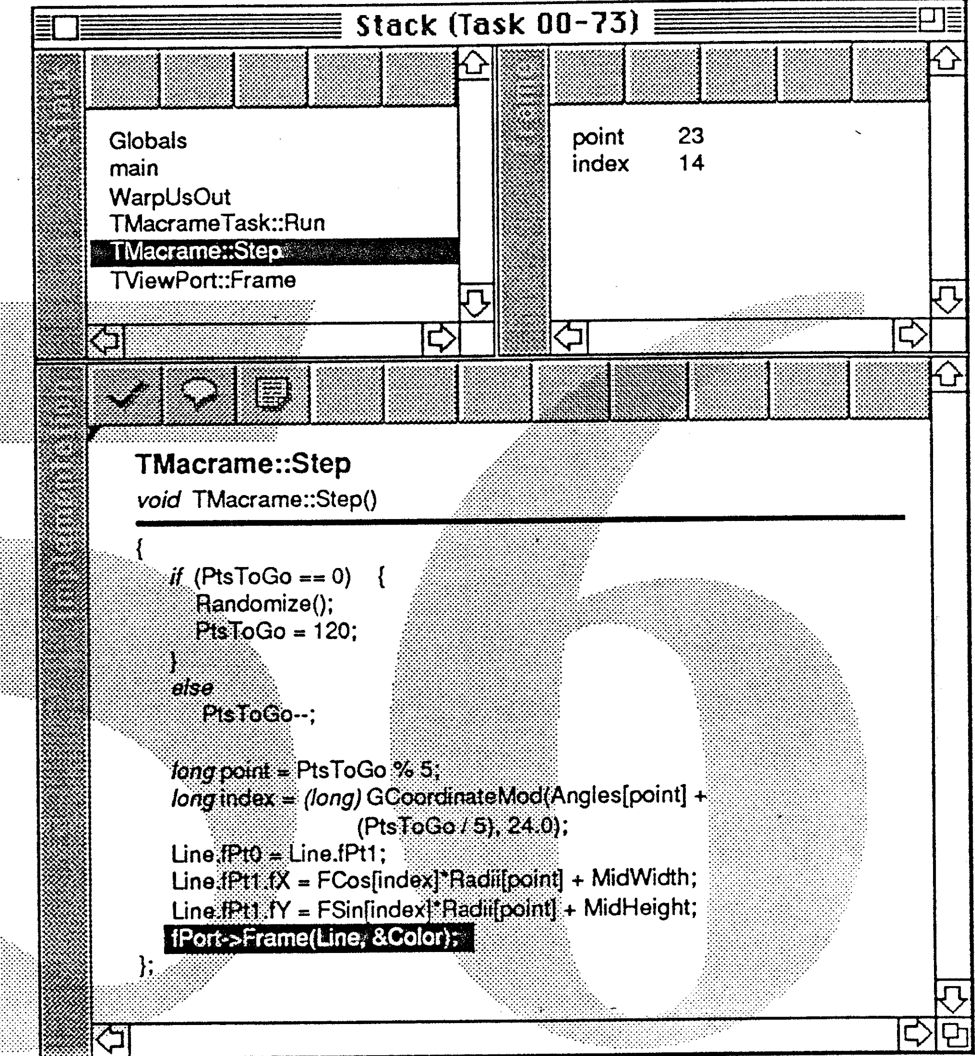


Figure 19. A stack browser. Separate panes display the stack, stack frame variables, and the source context for the selected stack frame.

Also under consideration is a technique which displays the values of scalar variables directly in the source adjacent to their use or declaration. Structured variables, such as C structs or objects, are displayed by selecting them or their reference in the source and then invoking a "Get Info" tool to show their values in a separate browser.

Still another method uses what we're calling a Data Connectivity Browser. This tool initially displays a browser with a rectangular graphic representing the stack. The graphic contains a vertical list of the current stack frames and can be

moved about and placed anywhere within the browser. A frame in the stack object can be double-clicked to expand in place, revealing the names and values of function parameters and local variables. Variables which are pointers can be double-clicked to produce additional rectangular graphics in the same browser, displaying the contents of the objects pointed at. A line with an arrow is drawn from the pointers to the new graphics showing the connections. All new graphics thus created can also be repositioned for a more pleasing display. Double-clicking a pointer that is currently displaying its contents removes that graphic (and any that it was pointing to) from the display. Appropriate navigating tools such as zoom in/out are provided to assist in dealing with large amounts of data displayed.

Control Flow

The hoops environment supports various types of breakpoints, stepping, and tracing. Breakpoints are set by pointing at the source location where the break is to occur. Our prototype shows setting breakpoints by selecting the line where the break is to occur and then clicking a stop sign icon in the icon bar. Special support is also available for setting multiple breakpoints, such as on all method calls, all methods by this name, etc.

Stepping through the source at some level of granularity is also supported. Visual highlighting indicates the current location. Clicking on a single-step icon causes the next increment to be executed. A command key is mapped to this function, however the command key mapping can be altered by the user.

The ability to suspend a task, modify a function, compile it, link it in, and then resume that task, is currently under investigation. There are serious difficulties in trying to provide this capability at any time during the program execution; therefore, there will likely be restrictions on how and when this feature may be used.

The environment also supports automatic tracing. Rather than the user needing to manually add entry and exit printing statements to all of his routines, the environment will either automatically generate calls to a tracing routine, or will force all calls to be indirect and then catch and report the call at the indirection point. A tool is also being considered for capturing this tracing information for a particular program execution and then graphically displaying a call/method invocation graph.

Debugging Language

The hoops system will not provide a separate debugging language. Actions such as inspecting, setting breakpoints, gathering statistics, etc. are performed using the mouse. Occasionally, however, a more sophisticated form of debugging, such as automatically taking some special action every time a breakpoint is hit, is useful. To achieve this type of functionality the user will write breakpoint action code in the source language. This code is compiled independently of the source, but in the same context of the breakpoint and executed when the breakpoint is hit.

Analysis Features

Beyond basic debugging features, hoops provides facilities for gathering and displaying additional program statistics. Some items currently being considered include the display of sizes, both source and code, frequency of execution information for detecting heavily used code, and instruction execution times over some range of source.

Graphical displays of system and program information will also be provided. Some of these displays may include team/task hierarchy diagram, heap usage, message traffic, performance information, etc. There are many other possibilities given the hoops architecture.

Machine-Level Access

Even with the greatest symbolic debugging facilities, it is still necessary at times to deal with the program at the machine level. hoops provides displaying of assembly language for any function and will allow the setting of breakpoints, stepping, etc., at this level as well. Displays of registers and uninterpreted memory will also be possible.

Remote Debugging

While the value of this feature is clearly understood, little work has been done in this area to date. Whether this capability exists in the initial release of hoops or not, some of the architecture to support remote debugging will be in place. In particular, for ease in porting to new hardware, a machine-independent interface will be developed for accessing memory, registers, etc. The implementation of this interface could also be modified to access this information over the network.

Multitasking Issues

A major complication to debugging in Pink is the expected use of multitasking in applications. The simpler problems include: selecting which task is being acted upon when inspecting, setting breakpoints, tracing, etc. The more difficult ones include: monitoring message traffic, controlling several executing tasks, deadlock situations, etc. Close cooperation with the operating system team is necessary to provide a debugging environment which deals with these problems cleanly.

Optimizing Compiler Issues

The Compiler Technology ANSI C/C++ compiler, and hopefully others, produces highly optimized code. In many cases this can make debugging at the source level more difficult than with a non-optimizing compiler. The problem lies in the fact that although a routine will be compiled to give the proper answer, the exact way it computes that answer may be significantly different from that described in the source language. Some things that the compiler might do include: eliminating code or variables known not to affect the final result, moving invariant code out of loops, combining common code, reusing registers allocated to variables when the variable is no longer needed, etc.

Mapping from source to code and vice versa can be difficult given some of the optimizations mentioned. One-to-many mappings are possible at times and hoops will need to inform the user when this is the case. Stepping though a routine may be on a larger than single statement granularity at times, and this will also have to be represented clearly.

Inspecting the values of variables, locals in particular, can be somewhat tricky also. One problem is that the value of the variable may not always be available at any location within the routine. In these cases hoops will inform the user of this fact rather than give incorrect information.

Modifying the values of variables in optimized code, again locals in particular, is especially problematic, and at times not possible. Unless specifically declared as volatile, the compiler "remembers" values assigned to variables and may use the "known" value later in the code without rereading the variable. A

change in that value "behind the compiler's back" could produce erroneous program results.

Fine granularity source level debugging with an optimizing compiler is generally an unsolved problem. A number of ideas are being considered for dealing with these problems. Disabling optimizations during development may alleviate some of the problems. Since hoops is an incremental system, another approach when encountering problems with the optimizer is to just add diagnostic code and recompile the function in question, especially if we are able to link the altered routine into the running program.

Constructor

What Is Constructor?

Constructor is the user interface design portion of hoops. It is used to create user interfaces for Pink applications (e.g. windows, menus, dialogs, alerts). In Pink, it provides many of the same capabilities that ResEdit and MacApp's ViewEdit provide for the Blue world. Constructor provides three essential capabilities within hoops: general resource editing, user interface layout and testing, and inter-object communication.

Within hoops, Constructor allows the user to edit icons, internationalize text messages, or to edit any other Pink resource.[47] In addition, Constructor is used for user interface layout. Examples include setting the on-screen location of windows, adding or rearranging controls within a dialog, or modifying menus or menu bars. With Constructor, users can create and manipulate library-like collections of resources called Parts Bins, which provide organized storage and retrieval of reusable resources and user interface elements. Finally, Constructor is used to set up simple inter-object communication paths by connecting together user interface elements.

hoops and Constructor provide for easy specialization or enhancement of existing resource types through subclassing (ie. specializing TView into TMyView so that it actually draws something useful). There is no real transition between manipulating object instances in Constructor and manipulating class definitions and hierarchies in the rest of hoops, due to Constructor's tight integration within hoops. Newly created classes are readily accessible to Constructor and can be added to a Parts Bin or incorporated into an existing application project.

What Constructor Is Not

There are many classes of problems that Constructor is simply not designed to address. The largest class is that of the "end-user programming" environment. Constructor is not a Pink Hypercard or SuperCard. Although Constructor provides many features similar to those found in end-user programming environments (a fact which arises naturally from our design goals), the user is required to have a deeper understanding of the Pink Toolbox and the hoops environment than can be expected of an end-user.

When creating an application's user interface, no source code will be produced by Constructor. This avoids a major problem encountered in current prototyping environments: once source code is generated, further changes must be made to either the source code or the prototype, *but not both*. Thus, Constructor should not be confused with "one-way" prototyping environments like Prototyper, AppMaker or even some aspects of the NeXT™ User Interface Builder which convert application prototypes into source code, but cannot propagate source code changes back into the prototype.

---

47. We consciously use the term resource here as a reminder that these entities are, in fact, the Pink equivalent of today's Blue resources. *Pink resources* are essentially just object instances stored on disk, or *persistent objects*.

The Foundation

Representation of Resource Elements

Many of hoops' powerful features are possible because the source code that makes up a project is understood to be a series of interconnected components and properties. hoops understands the *resources* (windows, dialogs, alerts, menus, icons, *etc.*) that make up an application's user interface[48] as a series of interconnected *resource components* and their associated properties. Resource components are a specialized type of hoops component designed to manage information about Pink resources.

Resource Components

Resource components share many characteristics with the generic hoops component, but in addition have several specialized properties. Resource components can represent either individual objects or collections of objects:

Atomic UI Components	Collection UI Components
Strings	
Icons	
Views	View Hierarchy (Window plus contents)
Menu Items	Menus, MenuBars
Controls	Control Clusters

Resource components can be further specialized into two additional component categories, *connectable components* and *user interface components*. Connectable components represent objects that either descend from MResponder (denoting that it can receive certain messages) or contain TOutputPorts (denoting that it can send messages in response to certain actions). User interface components represent objects that descend from TView and thus have some sort of on-screen representation. User interface components are implicitly connectable.

Constructor is designed to manipulate resources. Generic resource components can be inspected and their public data edited. In addition, Constructor allows connectable components to be connected together. Finally Constructor allows the on-screen location and appearance of user interface components to be modified and tested.

The resource components designed in Constructor are eventually stored as persistent object resources within a fork of a finished application. Either hoops or, more likely, the Pink Toolbox will provide some sort of "Resource Manager" for instantiating and flattening these persistent objects. This is analogous to the way MacApp (along with the Resource Manager) provides for instantiating and flattening view resources today. Often a set of related resources must be

---

48. User interface in a broad sense—essentially all of the flattened object equivalents to the "classic" resources in the Blue world.

instantiated together. Constructor can bundle together such resources into a *Resource Group*.

Resource Component Properties

Resource components have many of the same properties as code components (some with slightly different interpretations), plus several additional property types:

Generic Properties

Clients	Components that explicitly reference this object.
References	Components that are explicitly referenced by this object.
Container	Resource group containing this object.

Additional Properties

Interface	Information necessary to reference this object.
Flattened Object	The flattened object itself.
Icon	An icon used to represent the object in several of the views used by Constructor.
Actions	The list of conditions which trigger a connectable component object to send messages (plus the data type of each message).
Responses	The list of messages and their data types that a connectable component object understands.
Senders	Senders of messages for this object.
Targets	Targets of messages sent by this object.
Connections	The list of connections (message paths) originating from a connectable component object.
Superview	Superview, if any, of a view-derived component.
Subviews	List of subviews, if any, of a view-derived component.

Exactly how these properties are managed and manipulated will not be discussed in the ERS. However, the above information is presented to further elaborate h*oo*ps' underlying component model and how it is used by Constructor.

Presentation of Resource Elements

Constructor manages two different types of collections of resource data: *Resource Groups*, a collection of related resources typically found in applications and libraries, and *Parts Bins*, a library-like mechanism for organized storage and retrieval of reusable resource components.

Parts

Basically, a *part* is just another name for a Pink resource, both basic building block resources (e.g. icons, text messages, controls, scrollers, windows, menu items), as well as resource assemblies designed and saved by the user (e.g. OK button, standard Edit menu, Save Changes alert). We find it convenient to use the the term *part* when referring to the resources found in a Parts Bin, so as to distinguish them from the resources stored in an application under construction (although, in fact, the two may be identical).

Parts Bins

Parts Bins provide the user with a convenient metaphor for organizing, storing and retrieving parts. A Parts Bin is essentially a document that contains parts. The organization and manipulation of parts within a Parts Bin is discussed at length later in this document.

Constructor provides a default Parts Bin Browser (Figure 20), composed of a navigational view and a manipulation view. The navigational view contains a drawer representation of various parts categories. The manipulation view shows the contents of the drawer currently open.

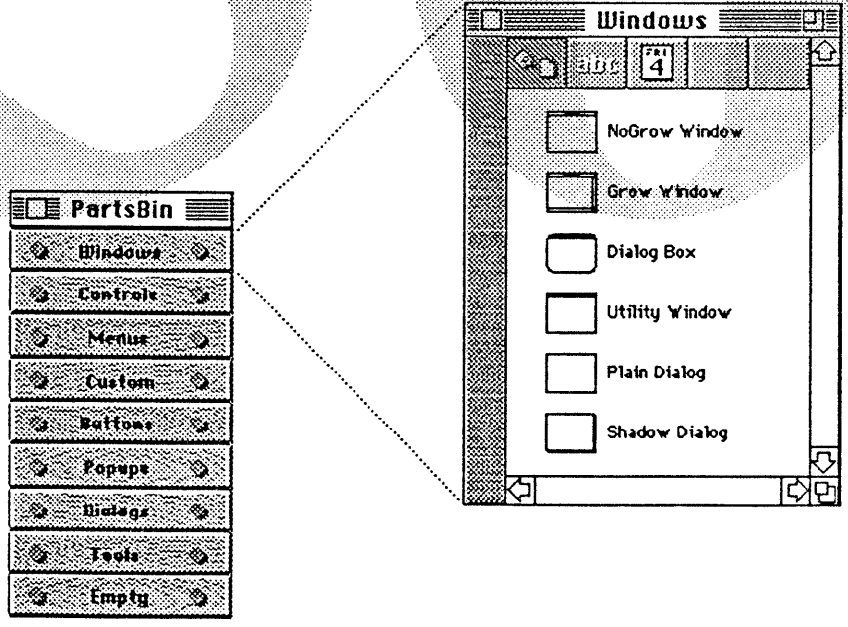


Figure 20. A Parts Bin browser and a drawer's contents.

h*oo*ps will ship with a Standard Parts Bin containing a useful set of Menus, Windows, Dialogs, Alerts, Controls, etc., so our users won't have to create their user interfaces from scratch. This standard Parts Bin will also aid in maintaining consistent user interface elements between applications (e.g. a consistent look for common dialogs such as "Page Setup", or whatever common dialogs are appropriate to Pink).

Resource Groups

Constructor also manages *Resource Groups*—collections of related resource components that are instantiated or revitalized as a group. Some applications will contain only a single Resource Group, while other applications will find it convenient or necessary to divide their resources across several Resource Groups. Groups may correspond to different document types, different phases of the application (e.g. printing vs. editing), or any convenient organization.

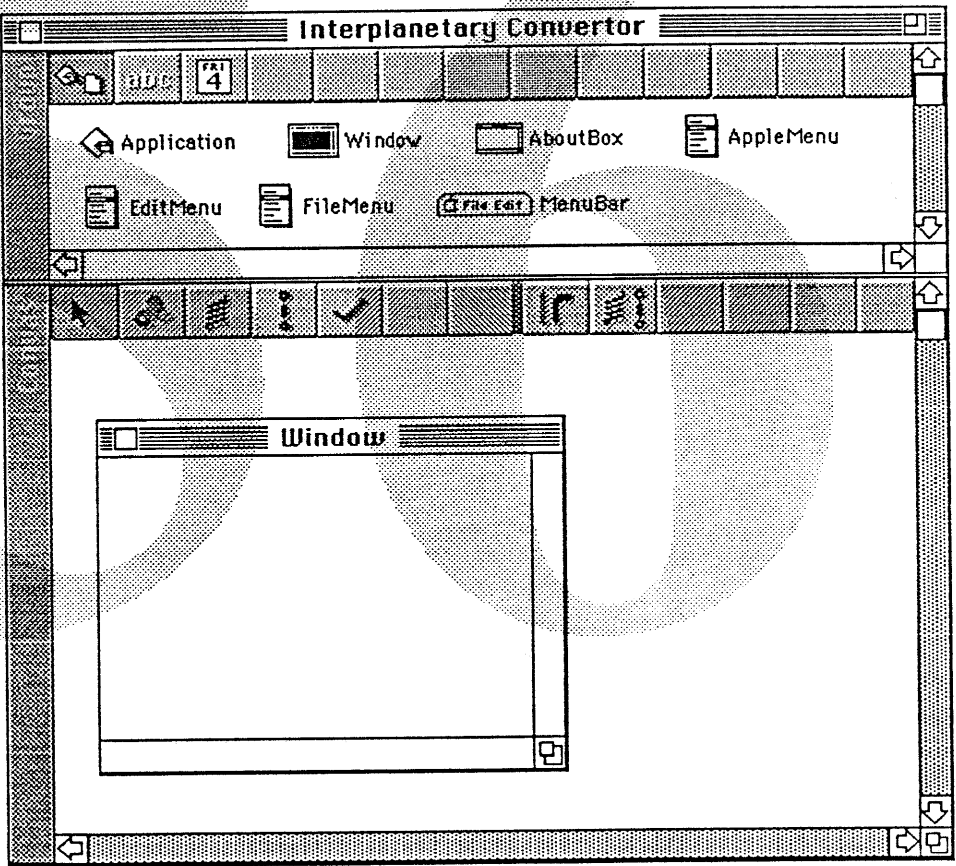


Figure 21. A resource group browser. The upper pane displays a view of all the resources in a group. The lower pane is where editing of those resources is accomplished.

Constructor provides a default Resource Group Browser (Figure 21) consisting of a navigational view (the upper pane) and an editing view (the lower pane).

The navigational view contains an iconic representation of all the highest level objects in the Resource Group (referring to menus, menu bars and windows— as opposed to the individual menu items and controls they will contain). The editing view contains the objects currently being inspected or manipulated. Which objects actually appear in the editing view at any given moment is determined by the item(s) selected in the navigational view. In the editing view, objects or collections of objects will appear and, as far as possible, function just as they will in the running application.

The application stationery included with hoops will contain one or more Resource Groups which can be used as a starting point for a new application's user interface.

Constructor Viewers

Constructor provides two standard viewers for dealing with Parts Bins: a *Cabinet viewer* and a *Drawer viewer*. In addition, Constructor provides two standard viewers for dealing with Resource Groups: a *Group viewer* and a *Canvas viewer*. These viewers are described in more detail in the Constructor Human Interface section.

A *Cabinet viewer* displays the various categories (Drawers) of parts the user has chosen to organize the parts in the Parts Bin (e.g. all Buttons vs. specific Buttons used in Print Dialogs). The number of Drawers, as well as their names and layout within the Parts Bin, is controlled by the user from within the Cabinet viewer.

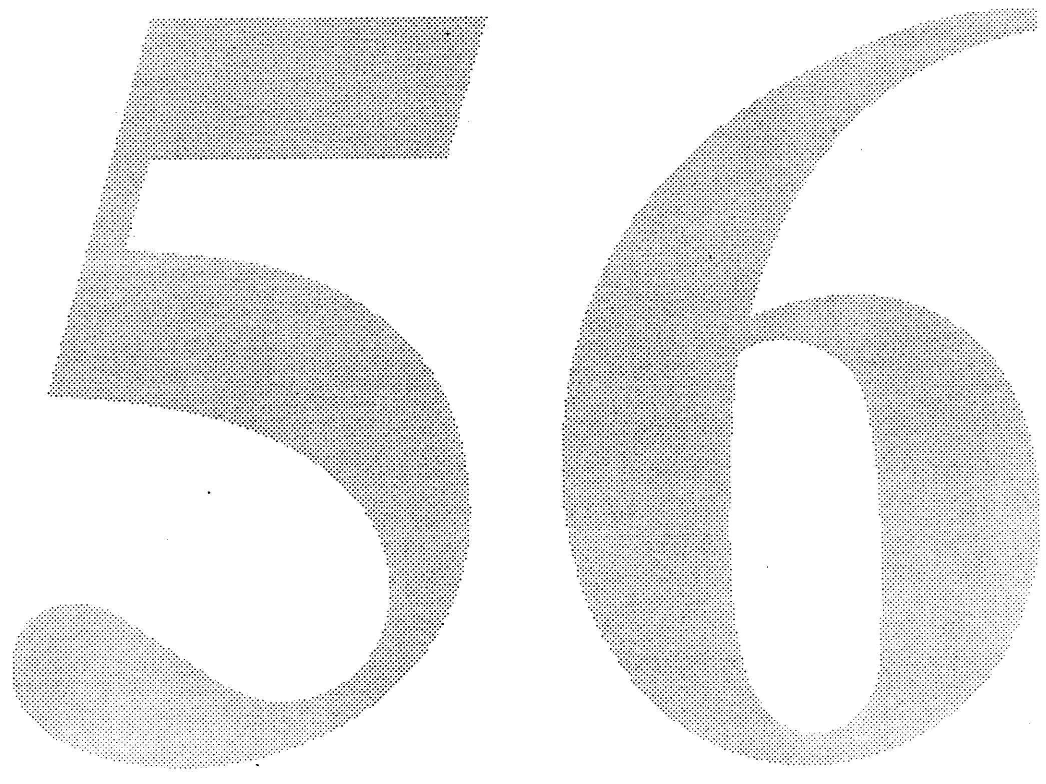
A *Drawer viewer* displays an iconified view of the parts in a particular Drawer. Drawer viewers are used when inserting, copying, or deleting parts in a Parts Bin. When adding a new resource component to an application under construction, a Drawer viewer serves as the source window.

A *Group viewer* displays an iconified view of the top-level resource components in a particular Resource Group. The resource components selected in the Group viewer are passed on as output to dependent views.

A *Canvas viewer* displays a life-size (perhaps scalable) view of one or more of the user interface components in a particular Resource Group. Most layout and editing operations take place within the confines of a Canvas viewer. To add new items, the user can drag the desired part from a Drawer viewer onto a Canvas viewer, or into another object visible in the Canvas viewer. As many of the editing and manipulation operations as possible will be available through direct manipulation (including size, location, font characteristics, titles, colors, menu item arrangement, etc.). Those operations not well suited to a direct manipulation interface will be available through some sort of object editor/inspector window, as is done in ViewEdit. Connections between objects (described later) are also made within a Canvas viewer.

Constructor is Always Available

Resource-related browsers are opened in the same way that any other hoops browser is opened. In addition, class components can be converted into Parts Bin parts with one of the selection-specific commands from the source viewers.



The Constructor Human Interface

In this section we show what a number of hoops viewers specific to Constructor might look like, and how they might operate. This set is intended to be representative rather than exhaustive.

Cabinet Viewer

A Cabinet viewer displays the drawers contained in a Parts Bin.

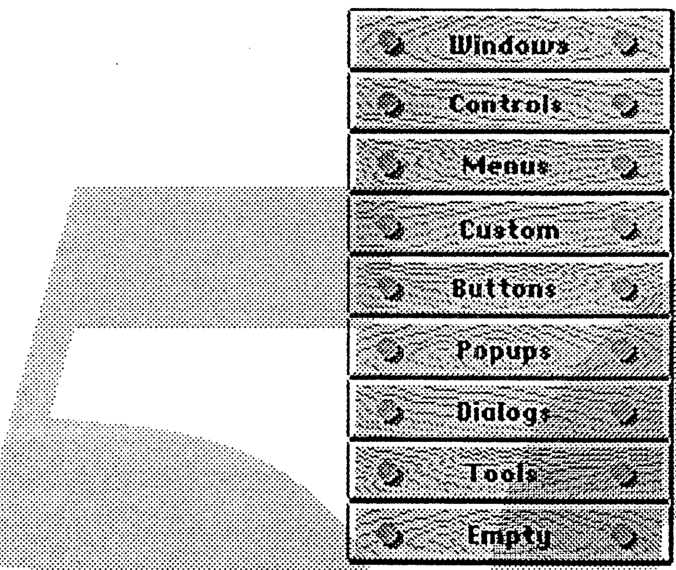


Figure 22. A cabinet viewer.

Purpose

Displays the categories of parts, represented as drawers, within a Parts Bin document and allows selection and manipulation of the drawers.

Input

A Parts Bin (default or from a library or project).

Output

Opened drawer or drawers.

Selection Techniques

Selecting a drawer name allows the name to be edited. Selecting one of the drawer pulls opens the drawer (sending it as an output). Selecting the body of a drawer allows the drawer to be repositioned within the cabinet.

Operations

Clicking a drawer pull, opens the drawer if closed or closes the drawer if already open.

An empty, untitled drawer is always present at the bottom (or right) of the Cabinet viewer. Renaming this drawer creates a new Untitled drawer. Drawers can be deleted with the Clear or Cut commands or with an as yet undetermined gesture.

Drawers are renamed by selecting their title and editing as in the Finder.

Drawers can be rearranged by dragging a drawer to a new location within the Cabinet viewer.

Drawers can be moved or copied between Parts Bins via Cut, Copy, and Paste, or via direct manipulation by dragging a drawer from one Cabinet viewer to another.

Dropping a part (resource component) onto a Drawer adds the part to this category (and if necessary copies it into the Parts Bin).

Resizing a Cabinet viewer forces the drawer positions to be reallocated to fit the new view size.

Drawer Viewer

A Drawer viewer displays the parts contained in a Parts Bin Drawer. Parts corresponding to both atomic components and collection components are shown. Individual components within a collection are not shown; instead, collection components can be manipulated as a whole. These parts are represented by small icons which are specified by the user.

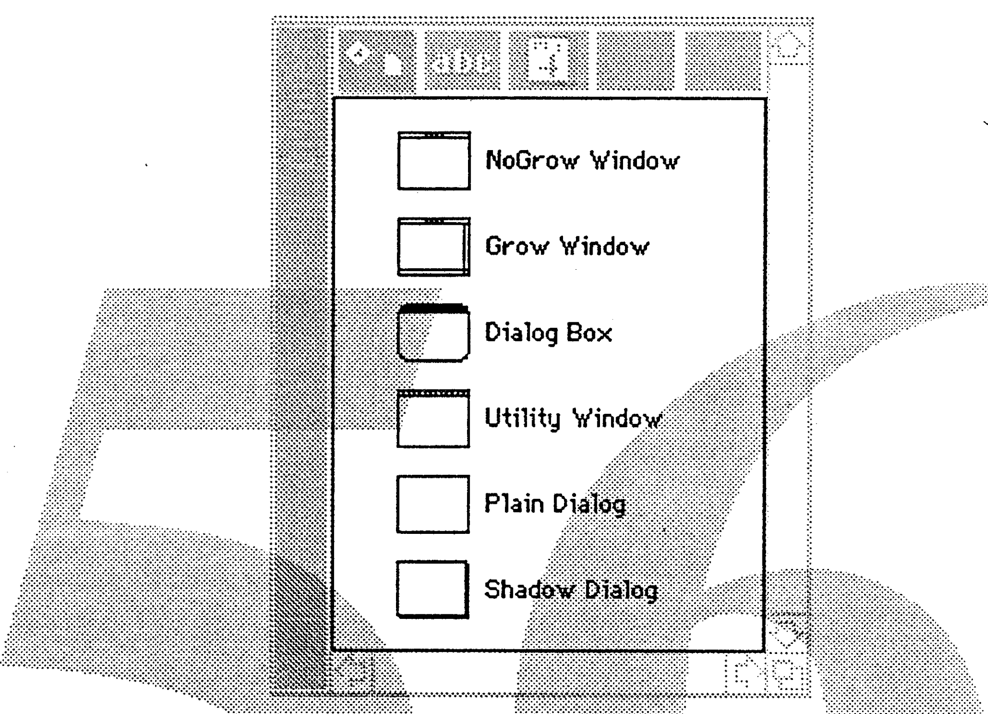


Figure 23. A drawer viewer.

Purpose

Displays the resource components, or parts, contained within a Parts Bin drawer. Each part is represented by a user-defined small icon and an associated title.

Input

A Parts Bin category (drawer).

Output

Selected part or parts (resource components).

Selection Techniques

Parts are selected by clicking their icons. Clicking the title of a part allows the title to be edited.

Operations

Parts can be moved or copied between drawers by dragging or via Cut, Copy and Paste. Similarly, parts can be copied between Parts Bins. Parts are added to a Resource Group by dragging the part onto the appropriate Group or Canvas viewer. Parts are renamed by selecting their titles and editing as in the Finder.

Class components can be converted into Parts Bin parts via a special menu command within a source code viewer. Comments or other descriptive information can also be associated with any of the parts in a Parts Bin, and can be viewed in a Description Viewer.

Group Viewer

A Group viewer displays the resource components contained in a Resource Group. Both atomic components and collection components are shown. Individual components within a collection are not shown; instead, collection components can are manipulated as a whole. Resource components are represented by small icons (copied from a Parts Bin along with the object).

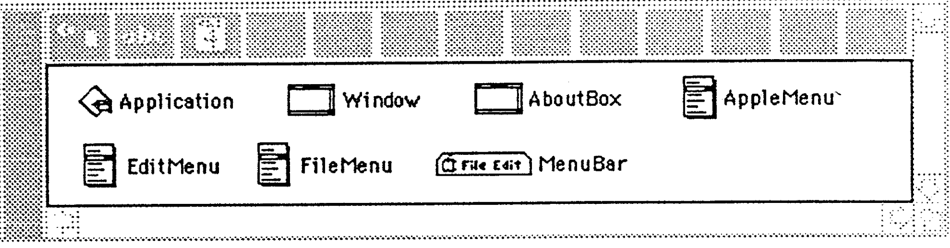


Figure 24. A group viewer.

Purpose Displays the "top-level" resource components contained within a Resource Group. Each part is represented by a small icon and an associated title.

Input A Resource Group.

Output Selected components (resource components).

Selection Techniques Components are selected by clicking their icons. Clicking the title allows the title to be edited.

Operations Rearranging items (positions are remembered).

Components are renamed by selecting their title and editing as in the Finder.

Resource components can be saved as parts in a Parts Bin by dragging them to the appropriate Parts Bin Browser views.

Resource components can be added to a Resource Group by dragging a part from the appropriate Parts Bin Drawer view into a Group viewer.

Comments or other descriptive information can be associated with any of the resources in a Resource Group, and can be viewed in a Description Viewer.

Canvas Viewer

A Canvas viewer displays a life-size (potentially scalable) view of one or more of the user interface components in a particular Resource Group. Most layout and editing operations take place within the confines of a Canvas viewer. Non-user interface components (non-view, like strings or icons) are displayed in iconic form.

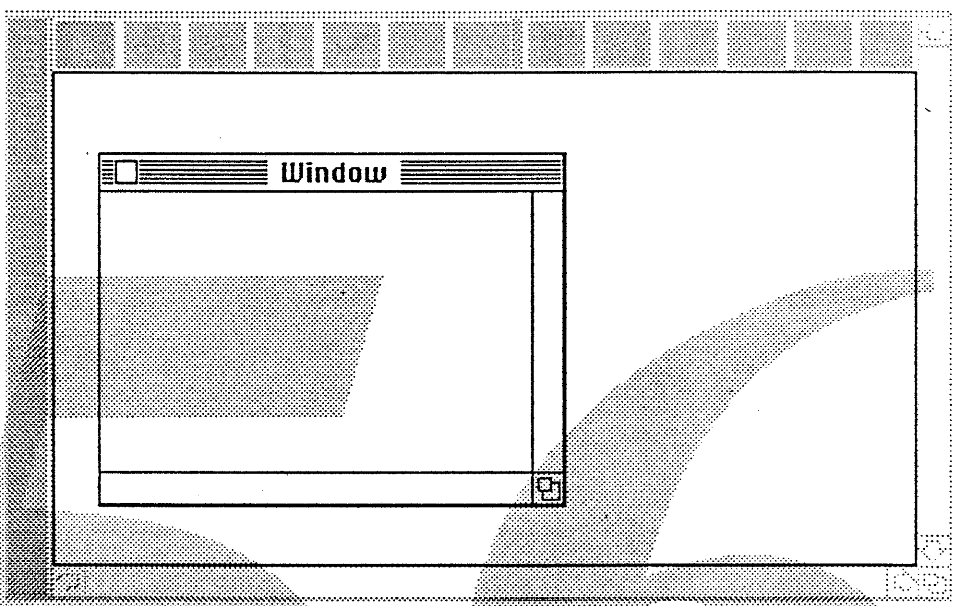


Figure 25. A canvas viewer. Editing operations on interface objects (such as the window in this case) are done within this viewer.

Purpose Displays a life-size view of one or more user interface components contained within a Resource Group.

Input One or more user interface components.

Output Selected components.

Selection Techniques Components are selected by clicking them as though in MacDraw. Double-clicking an object brings up an inspector for that object. Once an object is selected it can be moved, resized, or edited, as appropriate.

Operations

View Hierarchy Manipulation

Modify View Layout

View Editing/Inspecting

Menu Layout/Editing

Connection Editing

Resource components can be added to a Resource Group by dragging a part from the appropriate Parts Bin Drawer view into a Canvas viewer.

Resource components can be saved as parts in a Parts Bin by dragging them to the appropriate Parts Bin Browser views.

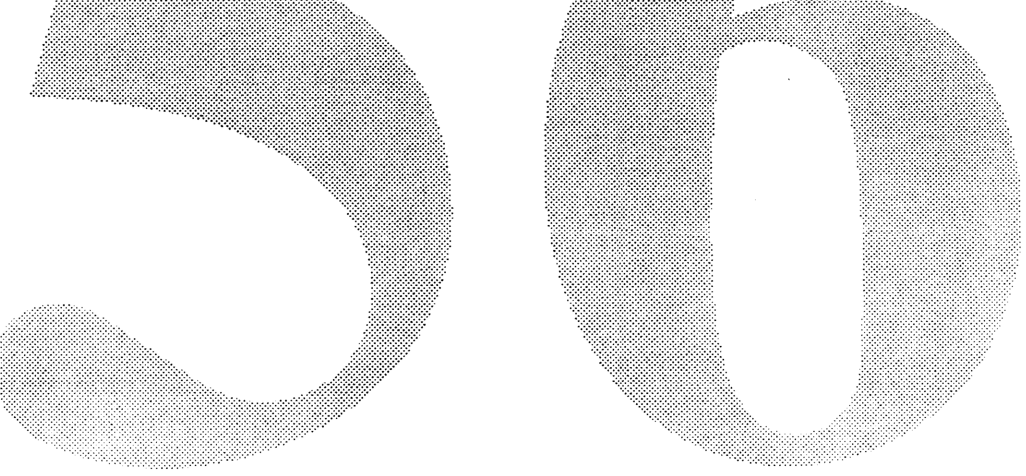
Tool/Icon Palette Functions

There are several different operations that can be performed within the Canvas viewer. including Edit, Constrain, Connect, Inspect and Test. There are also several layers (constraints and connections) that can be revealed. Switching between these operations will most likely be done through an icon bar associated with the Canvas viewer.

A User Scenario

Building a Celsius to Fahrenheit Converter

One of the hoops prototypes contains a mockup of Constructor which demonstrates how much of Constructor's functionality is used in building a simple temperature conversion application. Though somewhat dated with respect to our latest ideas, it conveys much of the "feel" of Constructor. If you're interested in seeing this mockup, please contact any member of the hoops team for a demonstration.



Constructor Details

Much of this section is currently incomplete or unspecified. Many of Constructor's capabilities are based on, as yet, unspecified features of the Pink Toolbox.

Extensibility

An essential aspect of Constructor is its ability to cope with new resource and user interface classes—classes created by the user, or supplied to the user by a third party. In order to perform its basic tasks, Constructor relies on the fact that each user interface class contains a set of basic editing methods (Draw, Grow, Move, Edit) or that this functionality is provided in some sort of "shadow class" whose name (i.e. TButton and its evil twin TButtonEditor) can be found in a dictionary maintained in the Parts Bin. These methods might be included at a sufficiently low level in the class hierarchy (TView) or as a default "shadow class", so that any new interface class will automatically inherit the minimal set of necessary behaviors (see "Object Editors" below).

Of course, we cannot guarantee that every class J. Random User creates can be used within Constructor. Classes which rely on external information (not including the view hierarchy) to draw or size themselves will fail. We will provide guidelines for "well behaved" classes designed for use with Constructor.

Many of the interface components present in the *resource* portion of an application will not be referenced directly by the code of the application since they will be resurrected from disk—essentially, instantiated by name. Thus it will be up to Constructor to make sure that the code for any non-standard (i.e. non-Pink) items is linked into the application. The mechanism for actually doing this will be defined when more of the Pink run-time becomes a reality. This impacts the inner workings of the Parts Bin, since the link between the flattened objects and the code needed to resurrect them would most naturally be stored with the component itself.

Prefabricated Parts

hoops will contain a standard Parts Bin full of ready-made parts, including standard menus, windows, alerts, etc. The exact contents and organization of this standard Parts Bin is yet to be determined.

hoops and Pink will also ship with various application templates or stationery. We envision that this stationery will include a standard set of resources and Resource Groups. Although the exact contents and organization of these resources is yet to be determined, we expect it to include at least the Application, some sort of document window, an about box, a menu bar, and several standard menus.

Canvas View Editing & Inspecting

Constructor also provides editing capabilities for resource components. For general resource components this will involve a series of resource type specific editors (strings, icons, fonts, etc.). For user interface components this will include setting font and color characteristics, adornments (a lá MacApp), window positioning, view resizing (with respect to its superview), etc. The poten-

tial complexity of this area becomes clear if one imagines combining the resource editors from ResEdit with the view editors from ViewEdit.

Editing General Resources

Constructor is able to edit essentially any type of resource that might be produced by the Pink system and applications. For common resource types (strings, icons, bitmaps, etc.) Constructor will contain a specific editor for each resource type. Unfamiliar resource types may either be edited in some generic raw (hex?) format or may be displayed read-only.

Editing User Interface Resources

Constructor will use a combination of two different approaches to allow editing of the internal properties of view-based or user interface resources. These are direct manipulation and inspector-like dialogs. Common editing functions such as setting the font or color characteristics will be performed in a familiar way without resorting to special dialogs. For example, to change the font size of any user interface item, the user simply selects the item and chooses the appropriate size from a familiar font or size menu. An additional advantage is that the change takes place immediately as opposed to when the editing dialog is dismissed. Many editing operations are either unique to an item or have no intuitive direct manipulation interface, so in addition we need some sort of item-specific editing/inspecting dialog, as in ViewEdit.

This editing dialog will appear either as a floating utility window or as a modeless dialog. The dialog will contain controls, text fields, etc. to allow one to modify all of the internal state of the item being edited (at least those fields accessible through its public interface). In an inheritance-based system the presentation of the accessible information poses a difficulty. The simplest form to implement separates the information into a series of panes corresponding to the various ancestors of the item being edited (as in ViewEdit). While simple to implement, this presentation suffers because typically the user is confronted with far too much information. In addition, partitioning of information with the ancestry of the object often leads to commonly modified attributes being scattered about the dialog.

As a solution to this problem, Constructor allows the editing dialog to have two forms, essentially a short and long form. In the long form the user is presented with total control over the object, whereas in the short form only the most commonly used (determined by decree) editing controls are available (Figure 26).

Object Editors

One, as yet unresolved, question is whether the actual code for an object editor lives in the object itself, or in a separate "shadow class". We currently believe

that objects will have to have some sort of "being-edited" state so they can respond differently to mouse-clicks, etc., when being edited. An example of these behaviors is that normally when the user clicks the title of a check box, it toggles the state of the box just as if the click had occurred in the box. In "editing-mode" however, we would like to have a click on the title of the check box mean that the title should be edited. This specialized knowledge about the internal structure of an item would seem to favor having the editing code defined as a part of the class itself. There are other issues to consider in deciding where this code should reside. If a "shadow-class" is used, Constructor must manage the extra baggage when copying parts from one Parts Bin to another. However, if the editing code lives in the class, then the author *must* provide the editing capability—it cannot be added or modified by the user.

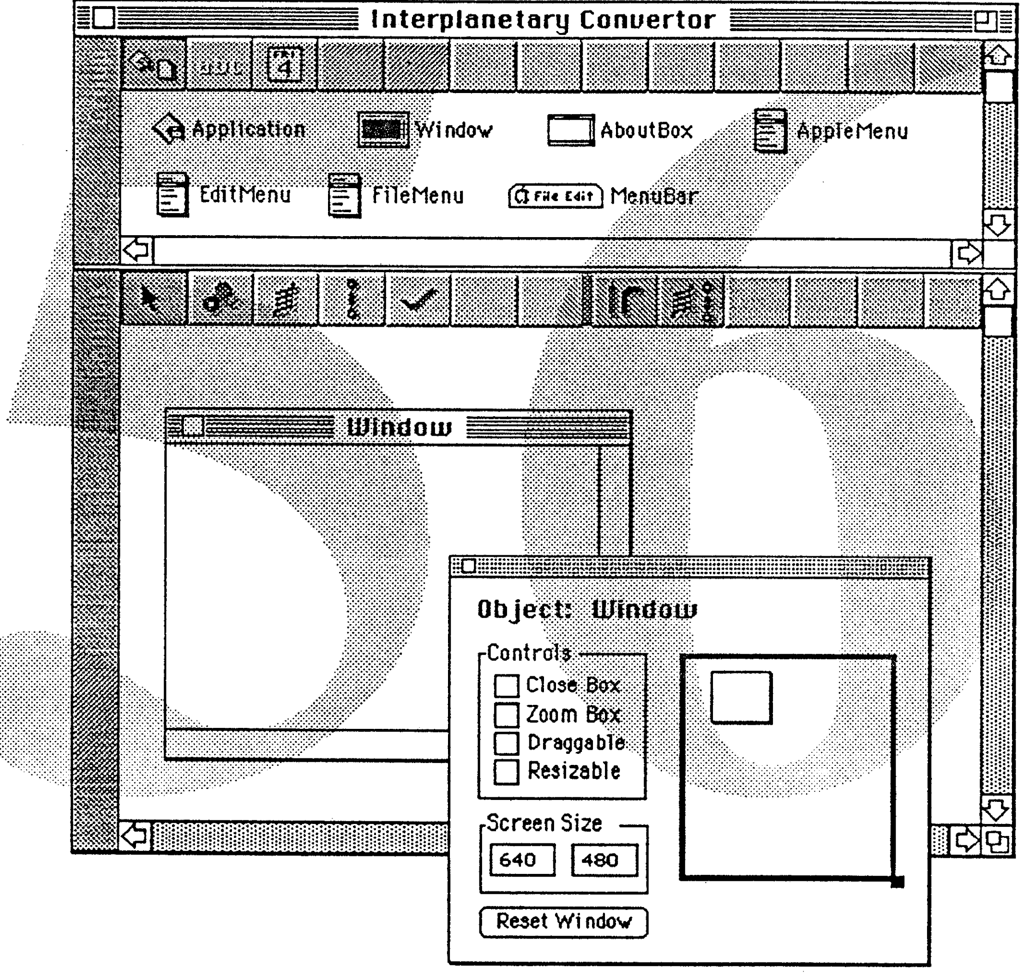


Figure 26. An example of an editing dialog for the window object displayed in the canvas viewer.

View Layout Details

Constructor will allow view location and size to be directly manipulated. In addition, the view hierarchy will be easily modified and traversed. The details of this are yet to be determined.

Constraint-Based View Resizing Details

The Pink Toolbox provides a powerful, but as yet undetermined, constraint-based view resizing mechanism. Constructor must provide a convenient user interface for specifying these constraints. The current Constructor mockups are based on springs, which allow for resizing, and fixed-size turnbuckles.[49] These constraints are attached between the edges of a view object and the corresponding edges of its superview.

Whatever model is adopted by the Pink Toolbox will be supported by Constructor. The effects of the constraints can be seen immediately when views are resized in the Canvas viewer. Constructor will also provide tools for screen layout and staggering of windows, including resizing based on screen size.

View/Menu Editing Details

Constructor provides for editing the internal state of resources. Whenever possible we will employ direct manipulation. Examples where this will be appropriate include setting of default text, fonts and colors. Editing of menu titles and menu item arrangment are also appropriate for direct manipulation.

Constructor will provide inspector dialogs to allow modifying attributes where direct manipulation is not appropriate, or in addition to the direct manipulation interface. As discussed previously, both long and short versions of these dialogs will be provided. The content and appearance of these dialogs has yet to be determined.

Connection Editing Details

To allow simple messages to be sent between objects, the Pink Toolbox provides Responders (objects capable of performing an action in response to a message) and Stimulators (objects that send a message to a Responder as the result of some internal state change—possibly triggered by the user). Constructor allows the user to make *connections* between such objects and to specify both the message trigger and its content. An example of making a con-

---

49. Similar to the model presented in *Building User Interfaces by Direct Manipulation*, Cardelli, L., Proc. ACM SIGGRAPH Symposium on User Interface Software, pp152-166, October 17-19, 1988.

nection can be seen in Color Figure 2 at the end of this document. This figure shows the connection inspector window, which is used to view or modify connections.

Connections

This section explains the connections between objects produced by Constructor, as well as the gory details of the underlying source code. At present, most of what follows is unimplemented in Pink. In several discussions with David Goldsmith and Frank Leahy we have agreed on this general architecture as satisfying the needs of both Pink and Constructor.

Responders

In the current Pink architecture, objects that can respond to messages (text or token based—as opposed to actual method calls) are created with the MResponder mix-in. When creating such a class, the programmer specifies a dictionary of messages that the class understands and the class method that handles the message. Each Responder instance can add additional messages to an instance-specific response dictionary. (In the current Pink architecture these message handlers are limited to class methods, as opposed to script or arbitrary code.) The methods designed to handle messages are all of the form:

```
Boolean HandleSomeMessage(TStimulus* msg);
```

TStimulus is a container for the tokenized message text. Subclassing TStimulus allows data to be packaged with the message. This allows any object to send a message to any other object, regardless of the message contents, with a single method call (e.g. TBooleanStimulus vs. TFloatStimulus vs. TTextStimulus):

```
targetObject->Respond(TTextStimulus("SetText","mytext"));
```

Of course since the responder expects a TTextStimulus to be sent with the "SetText" message, strange things will happen if a TBooleanStimulus is sent instead.

Stimulators

Responders are only half the picture though. For Constructor, we need an equally flexible way to have objects *send* messages to arbitrary targets in response to some internal state change. For instance when an OK Button is clicked it should be able to send a message. In MacApp this message is sent up the view hierarchy, via the method DoChoice(), to the document and finally to the application. Somewhere along the way someone will take an appropriate action. While sufficient for many situations, this architecture does not support the communication necessary for the style of user interface programming Constructor needs to support (e.g. dependency graphs). Code must be written to implement the message network.

What we need is the following: each user interface component has a predetermined set of conditions under which it may send a message to another object.

Examples of these conditions include: being clicked, on a keystroke, while the mouse is over, and various types of state changes (e.g. from empty to containing text, or dimmed to highlighted). Let's call all objects with this capability *Stimulators*, since objects other than user interface components may want this capability as well.

OutputPorts

In the definition of a Stimulator, there will be a list or array of OutputPorts, one for each of the conditions under which the object will want to send a message. An OutputPort contains a pointer to the target for the message, if any, which must be derived from MResponder, and a pointer-to-member-function which points to the specific method in the target object that should handle the message.

Here's what a TOutputPort might look like:

```
typedef Boolean (MResponder::*ActionPMF)(const TStimulus& msg);

class TOutputPort : public ... {
     MResponder*    fTarget;
     ActionPMF   fAction;                         // PtrToMember decl

     Send(const TStimulus& msg) {                 // Illustrative only
         if (fTarget != NIL)
             (fTarget->*fAction)(msg);            // PtrToMember call
         }
     }
};
```

Here's what a TButton might look like, if it has two OutputPorts, one for while the mouse is pressed, and one for when the mouse is released in the button:

```
class TButton : public TView {
      TOutputPort  fClickedOutput;
      TOutputPort  fStillDownOutput;

      HandleMouseStuff() {                        // Illustrative only!
          while (StillDown()) {
              TBooleanStimulus msg(MouseOverButton?);
              fStillDownOutput.Send(msg);
          }
          if (ReleasedOverButton()) {
              TStimulus msg;
              fClickedOutput.Send(msg);
          }
      }
};
```

Pointer-to-members are used, instead of the usual token-based messages for increased performance.[50]

---

50. This pointer-to-member based implementation is not required. Token-based messages could be used if performance is not an issue.

Connecting Objects with Constructor

When the user draws a connection between two objects (described in the *Canvas Viewer* Human Interface section above), Constructor will have to check that the message sender is in fact a Stimulator and that the target object is a Responder. In addition to message source and destination, the connections, or rather the messages sent via the connection, have an associated type. Using the message type, Constructor can make certain that OutputPorts which send a message of type TTextStimulus (or a subclass) are only ever connected to message handlers that expect to receive messages of type TTextStimulus (or a superclass). The names and types of the sender's TOutputPorts and the names and types of the target's message handlers will be maintained by hoops since these are *properties* of connectable components. Constructor can enforce this type compatibility across the connections by disabling incompatible choices (see the Connection Inspector Window in Color Figure 2 at the end of this document).

When the user completes a connection, Constructor then fills in the appropriate TOutputPort in the sender side of the connection with a pointer to the target and the appropriate target member function pointer.

A Connection Toolkit

The Connection Toolkit provides a small set of useful objects for building up more complex connections, for instance, dependency information between dialog items. This set of might include inverters, gates, splitters, converters, repeaters, etc.

Dependencies on Pink

hoops is a Pink Application

It is important to remember that hoops will be one of the first, and possibly for quite a long time, one of the largest Pink applications. This means that hoops is critically dependent on parts of Pink being ready and available, and capable of handling certain loads.[51] The development environment we want would take much longer and require much greater manpower to accomplish, if the entire program was being implemented from scratch. Fortunately this is far from the case because hoops is being implemented on and in Pink.[52]

People associated with Pink have always been flexible and willing to help. In fact we would like to think that all of Pink is engaged in the creation of hoops. This doesn't mean that anyone designing Pink functionality needs to alter their plans.[53] All we ask is that if you are designing Pink functionality, you remind yourselves that you are building your own working environment, and for that reason it should be as fast, as flexible, as easy to use, and as beautiful as possible.

Particularly Important Dependencies

Here is a list of areas where we expect to make heavy use of standard Pink capabilities. This means that in these areas, expect hoops to have essentially the capabilities and behavior of any other well-written Pink application. Any capabilities built into these areas of the Pink toolbox will be capabilities in hoops, and conversely any limitations of these areas will be limitations in hoops. This list is certainly incomplete but should indicate how dependent hoops is on the rest of Pink.

Text

hoops will use the system-supplied text services for editing, with only very specialized overriding of standard facilities. We expect to use ZZText and Jane capabilities pretty much as delivered. But remember that this means unlimited text, with styles, graphics and international support and so on.

Versions

We expect to use the versioning facilities supplied by Pink. The Program management chapter of this document describes the sort of capability we think we

---

51. To provide the right degree of contrast, try to imagine MPW editing built on top of Blue TextEdit.
52. hoops will be used to implement hoops from the earliest possible opportunity.
53. Pinkos, don't take this as a promise that hoops won't cause you to change both your working habits, and your designs. This will most certainly happen, but always bearing in mind our common goal of producing a working environment that we ourselves, as designers, artists, authors, and programmers would want. We apologize if this sounds like hype. Ideals need to be lofty so that they do not chop the heads off results.

need, but we are flexible in this area and believe that a powerful, standard scheme is important.

Collaboration

hoops will follow the Pink model of collaboration and we expect most of the hard design and implementation work will be done by Pink.[54] There is a limited description of what we think we need in the Program management chapter. A very important thing (remembering that "us means you"), is to be able to take our work home.

Undo

The standard model is what us software folks get.

Hypertext

hoops needs it. We would rather use the standard Pink model even if we have to write our own implementation of functionality.

Document and Application Architectures

Not only does hoops need to know about them, it intends to use them.

Scripting

hoops will make all appropriate access to its internals available through the standard Pink scripting facility.

User Interface

We are particularly dependent here since our user interface is being designed alongside the rest of Pink's. hoops is a very early Pink application but we still want it to exemplify the Pink human interface.

Finder and File system

Like every other Pink application, hoops will be dependent on the Finder and the File system. However hoops requires special services with regard to launching applications, among other things.

---

54. Thanks Arn, Larry, Jack, and Mrs. Calabash (wherever you are).

Dynamic libraries

These are crucial. The design and extensibility of hoops depends on them.

OS and Network

In order to debug programs in the most powerful and straightforward way, hoops will need certain run-time and network facilities.

Resource Manager

hoops relies on the Pink Toolbox's ability to save and restore flattened objects with embedded pointers. This also includes the ability to flatten pointers to member functions. hoops needs some sort of structure imposed on the file's fork that contains flattened objects. It is necessary to resurrect objects by name (or by id), similar to the way the Resource Manager works today.

Editing User Interface Objects

hoops will need some toolbox support so that objects can be both actuated and edited in the Canvas viewer. Not only do objects need to know how to be dragged/resized/renamed, but they need to support some sort of a general mechanism for viewing and modifying specific attributes.

Constraints

Currently, hoops uses a "springs and turnbuckles" model to impose constraints on user interface objects. These constraints are *rules* for how an object should be resized when its superview changes size. hoops and the ToolBox team need to work together to implement support for a view-resizing model.

Dependencies on CompTech

Who Said Anything About Compilers?

The observant reader will have noticed that this document assumes the existence of at least one compiler. The observant and astute reader will have noticed that this compiler is being asked to do some mildly unusual things.

It almost goes without saying that hoops needs the compiler(s) developed by the Compiler Technology team. In order to provide the incremental system envisaged, it is necessary for the compilers and the rest of the environment to communicate much more fully than has been usual in the past. Language sensitivity in the editor for example, will rely on information created by the compiler. Conversely the compiler will rely on symbol and dependency information maintained by the hoops environment.

There are many issues to be covered here that will require the active and enthusiastic involvement of the CompTech team.

Glossary

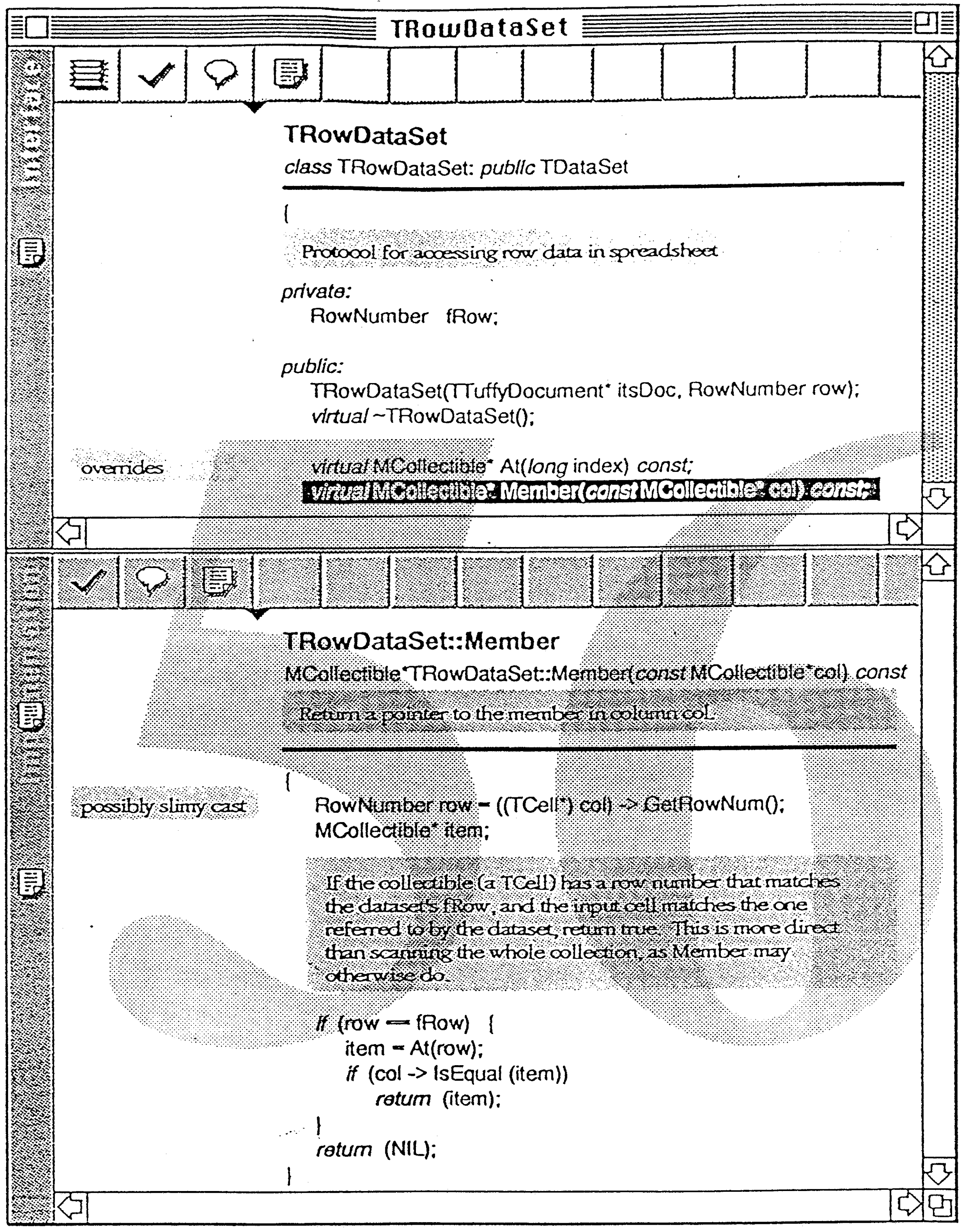
ATOMIC COMPONENT	a component that is not further subdivided into smaller components.
BROWSER	a loosely used term to describe a window which has two or more panes.
BROWSER CONSTRUCTION KIT	an archaic term used to describe the technique for creating views that interact with each other, or "connect" to each other.
C++	a really swell programming language.
CABINET VIEWER	a view of drawers containing the resources in the parts bin.
CANVAS VIEWER	the Constructor viewer that displays a life-size view of one or more of the resource components in a particular resource group. The canvas is Constructor's work area for creating new resource components and wiring connections between connectable components.
CLASS COMPONENT	a collection component that represents a class in object-oriented programming.
CODE COMPONENT	a component that represents some part of a program that is written in a programming language.
COLLABORATION	a Pink technique that we intend to use to support team programming. Collaboration allows for multiple users to "simultaneously" access the same document, or different parts of it.
COLLECTION COMPONENT	a component which has a property that enumerates other components. An example is a library, whose primary purpose is to represent a collection of other components.
COMMENT BLOCK	a single, connected set of word-wrapped lines. A comment block can be marginal, private, or public (i.e., a description). See also marginal comment, private comment block, and description.
COMPONENT	a named object that represents a semantic element of a program. All data in a project is represented as components. Examples of components include classes, functions, type definitions, and libraries. See also atomic component, code component, collection component, connectable component, organizational component, resource component, and user interface component.
CONFIGURATION	a set of versions of components that make up the state of a project at a given time.

CONNECTABLE COMPONENT	a resource component that descends from MResponder and/or MStimulator and has the ability to send and/or receive message to/from other connectable components.
CONNECTION	a metaphor that represents the trigger and message sent from one connectable component to another.
CONSTRUCTOR	that part of h*oo*ps used to create the resources of a program, most importantly, the user interface of a Pink program. See also canvas viewer, cabinet viewer, drawer viewer, resource group, parts bin, and resources.
CUSTOMIZATION	the act of creating new and unique combinations of the existing set of tools in h*oo*ps. See also extension.
DAVE AND TOM	see Tom and Dave.
DERIVED PROPERTY	a property whose value is not stored as part of a component. Rather, it is derived from other properties of the same or other components.
DESCRIPTION	a property of a component that acts as a public comment block describing a component. Descriptions form the basis of an on-line documentation system for program components.
DRAWER VIEWER	a view of the contents of a cabinet drawer showing the available resources. Generally the parts in a drawer will all share some common classification. The *Controls* drawer, for example, may contain a radio button, check box, OK button, scrollbar, etc. See also cabinet viewer.
DYNAMIC CLASS	a feature of the Pink run-time architecture that permits classes to be added to and used by an existing program, without relinking the program in the traditional sense. (Some form of run-time linking must occur for the new classes to be accessible to the program.)
EDITOR	a view that permits editing of its data via the combination of a viewer and one or more transformers.
ENVIRONMENT FRAMEWORK	the interface into h*oo*ps itself, much like the Pink Application Framework. It provides the classes and definitions from which new tools can be created and h*oo*ps extended.
EXPORT	the act of converting one or more h*oo*ps components into a form external from a project, such as an operating system file.
EXTENSION	the act of adding new and unique tools to h*oo*ps.
FILE COMPONENT	a component that represents a stream of text, as in an MPW text file, though it is not saved as an operating system file.
GROUP VIEWER	a view of the top-level resources in a particular resource group.

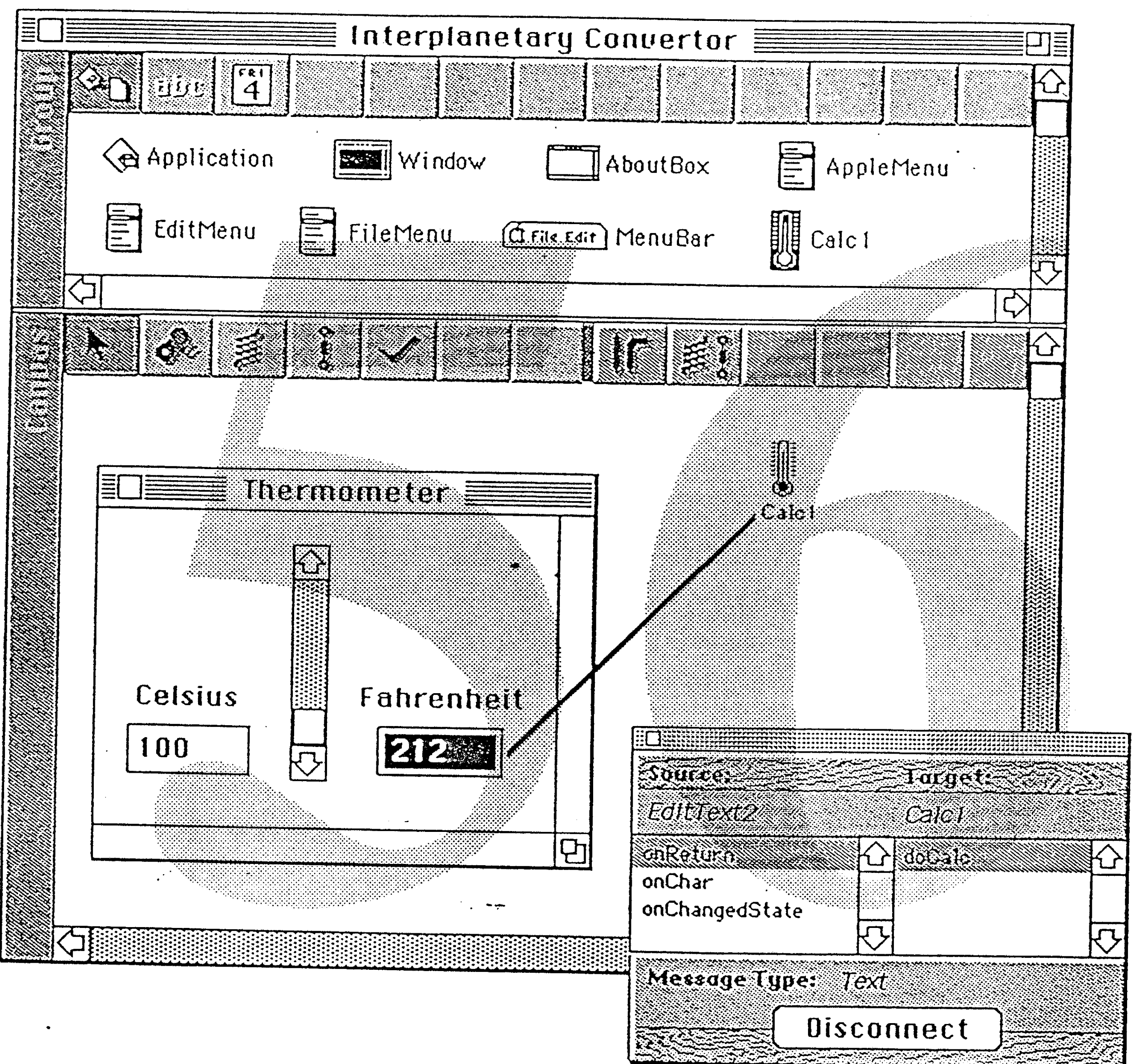
HOOPS	human-oriented object programming system. Sometimes incorrectly referred to as the heretical object-oriented programming system.
IMPORT	the act of converting source code from an external source into one or more hoops components.
INTRINSIC PROPERTY	a property whose value is stored as part of a component.
MACRO	mutants to be avoided at all costs.
MARGINAL COMMENT	a comment placed in the margin area of source code.
MESSAGE	a Pink message (TStimulus) that contains data to be sent between resource components in response to a stimulus.
MODULE	a component used to group non-object-oriented atomic components (e.g., functions, type definitions, constants, variables, etc). It is roughly equivalent to the class for non-object-oriented code.
NAVIGATION	the act of "traversing" through the components of a project.
ORGANIZATIONAL COMPONENT	a component whose purpose is to arrange other components into a group.
OUTPUT PORT	a connectable component's trigger that initiates a message to be sent to another object. A scrollbar, for example, may have scrollUp, scrollDown, pageUp, pageDown, and valueChanged output ports. See also stimulator.
PANE	a rectangular subdivision of a window, capable of displaying a viewer.
PART	a reusable interface item such as a check box, scroller, or a Save Changes dialog.
PARTS BIN	used for organizing, classifying, storing, and retrieving resources for Constructor.
PRIVATE COMMENT BLOCK	a comment block embedded in the implementation of a component.
PROJECT	the source code, object code, documentation and related data of a software development effort, represented as a wholly independent entity to hoops.
PROPERTY	a characteristic or attribute of a component. Components can have any number of properties, depending on the component's type. Examples of properties are source code, object code, interface, or description. See also derived property and intrinsic property.
PROTOCOL	the set of member functions of a class that defines how other parts of a program communicate with a class and its subclasses. By having a protocol, it is possible to add new subclasses without changing the other parts of the program, because the subclasses respect the protocol.

56

RESOURCE	a flattened object stored in a fork of an application.
RESOURCE COMPONENT	the h*oo*ps representation of a resource.
RESOURCE GROUP	a collection of related resources that are instantiated or revitalized as a group.
SELECTION-SPECIFIC MENUS	A set of commands or operations that apply specifically to a given selection in a view. One example of a selection-specific menu is the pop-up menu used in the current hoops mock-up.
STIMULATOR	a connectable component that sends a message in response to a state change (i.e. a character was typed, the button was clicked, etc.). See also output port.
STYLE SHEET	a technique for associating a visual style with the syntactic elements of source code.
TOM AND DAVE	two men with brains (or is it egos?) the size of planets.
TOOL	an object capable of performing some action, or displaying some data in h*oo*ps. h*oo*ps is implemented as a set of tools. Note that this definition deviates from, and is much broader than the MPW definition of a tool.
TRANSFORMER	a tool that changes the value of a property of a component.
TRANSLATOR	a tool that transforms a property of a component into another property of the same or another component.
USER INTERFACE COMPONENT	a connectable resource that descends from TView used in constructing the user interface of a Pink program. These objects include windows, menus, dialogs, alerts, and icons.
VERSION	any saved state of a component.
VIEW CONSTRAINTS	a Constructor model using springs and turnbuckles and associated rules for how an object should be behave when its superview changes size.
VIEWEDIT	a Blue tool for creating MacApp view resources used in a MacApp program.
VIEWER	a tool that displays a property of a component.



Color Figure 1. Source code browser. This browser consists of two connected views: an interface view and an implementation view. Comments are shown with a gray background.



Color Figure 2. "Wiring" a connection between two objects in Constructor's Canvas view.

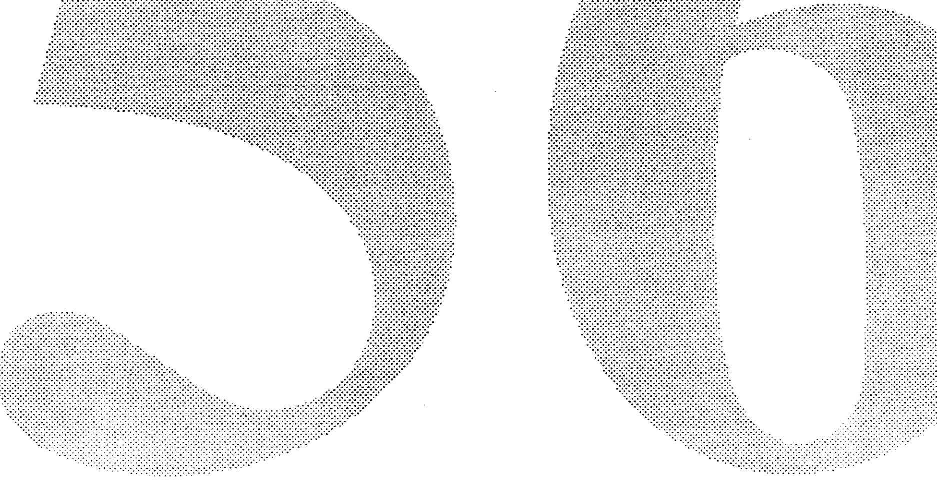
56 Online Documentation

56

Online Technical Documentation for Pink

(a plan)

by Scott Knaster, March 15, 1990



56

General plan

The Developer Technical Publications group, reporting to Trish Eastman, and Scott Knaster, reporting to Doug Brent, are working on creating technical documentation that Pink developers will use to write software. A large part of this documentation will exist online and will be available to developers in the "heat of battle" – that is, directly from HOOPS, the Pink development system. This paper describes the preliminary plan for putting Pink technical information online.

The complete Pink technical documentation suite is described in another document in this binder, but here's a brief description of the parts. There will be an overview of Pink capabilities, designed for people who will evaluate whether to use Pink. We expect this will look something like the Understanding Computer Networks book published last year: lots of pictures, a friendly level of technical detail.

There will be two books designed to introduce Pink in more depth, one for those who like to read about theory before starting to program (working title is The Pink Foundation), and a more tutorial volume for those who like to learn as they go (How to Write a Pink Program). We will investigate producing interactive online versions of these books.

The documentation will include an online reference to every Pink class and member function, tightly integrated with HOOPS. This online reference makes up the majority of the project described in this document.

The final part of the (paper) documentation suite is a series of focus books, each of which focuses on a specific Pink topic, such as graphics, imaging, text, views, and so on. Each focus book will be the definitive reference for its topic.

Goals and scope of the project

In this project we envision two kinds of online documentation: reference descriptions of classes and members (which we call simply **descriptions**), and narrative documents which include hypertext links, content browsing, multimedia, and other nifty Pink features (which we call **online books**).

The online descriptions are the heart of our project. We've already started writing them. Eventually, they'll be displayed in HOOPS (and maybe elsewhere, as well). Until HOOPS is born, you'll be able to read online descriptions with Mouser, 411, and HyperCard. The appearance of online descriptions in HOOPS depends on the design of HOOPS itself, and probably the Component Editor in particular (if that's still a valid name for a place in HOOPS).

Online books are the more interesting (= harder) part of the project. Officially, we're just investigating this part of the project. That's because we'll have paper versions of the documents to fall back on in case we can't complete a online book in time. As a minimum, we expect to have online a version of each book that's functionally the same as the paper version.

Our idea for online books is this: imagine HyperCard, except that (1) the metaphor is not an index card, but a book, and (2) it's done in the Pink application engine, so it has CHER and everything else that's Pink. We think that using a book metaphor is a good idea. Everybody knows how to use books in a linear fashion, and many people know how to use tables of contents and indices – sort of a "manual hypertext". A book provides a comfortable, linear frame of reference for users, which they can use sometimes (by reading linearly) and abandon other times (by following links).

Our online books will look like books on the screen, complete with (optional) page numbers. But they'll be magical books, because they'll include hyperlinks, sound, animation, content browsing, fold-out pictures, annotation, and other Pink goodies. Of course, online books can be also be used in a completely linear fashion, and can even be turned into paper books automatically by removing the multimedia and hypertext features. In fact, existing OCR technology could provide developers with the ability to automatically create online books out of paper books (although without any hypertext or multimedia additions, of course). The developer could then integrate the online magic.

We believe that the book metaphor provides a rich foundation for placing information online and presents a compelling framework for users. Of course, we don't really know how these ideas for online books will work out, but we'd like to try them. Serendipitously enough, the Jane group is interested in trying some of the same things.

Survey of existing systems

To help make our system great, we'd like to try standing on the shoulders of giants, or at least their kneecaps. If somebody else has done something great, we'd like to gain inspiration from it (ATTENTION RABID LAWYERS: as long as we can legally do so, of course), and if somebody else has discovered some blind alleys, we'd like to know about that, too. This means a thorough investigation of existing systems is in order. For a list of the systems that we've looked at, or to make suggestions for the list, talk to Scott Knaster.

Create, test, and evaluate prototypes

After we've gathered a good collection of ideas, we'll create prototypes to see if those ideas make any sense. We think we can use HyperCard 2.0 (now in beta release) as a platform for these experiments. Because we're writing online descriptions in 411 format, we'll be able to display them in HyperCard via a 411-reader XCMD that exists now. We also may be able to integrate our online description prototypes into the HOOPS SuperCard prototype.

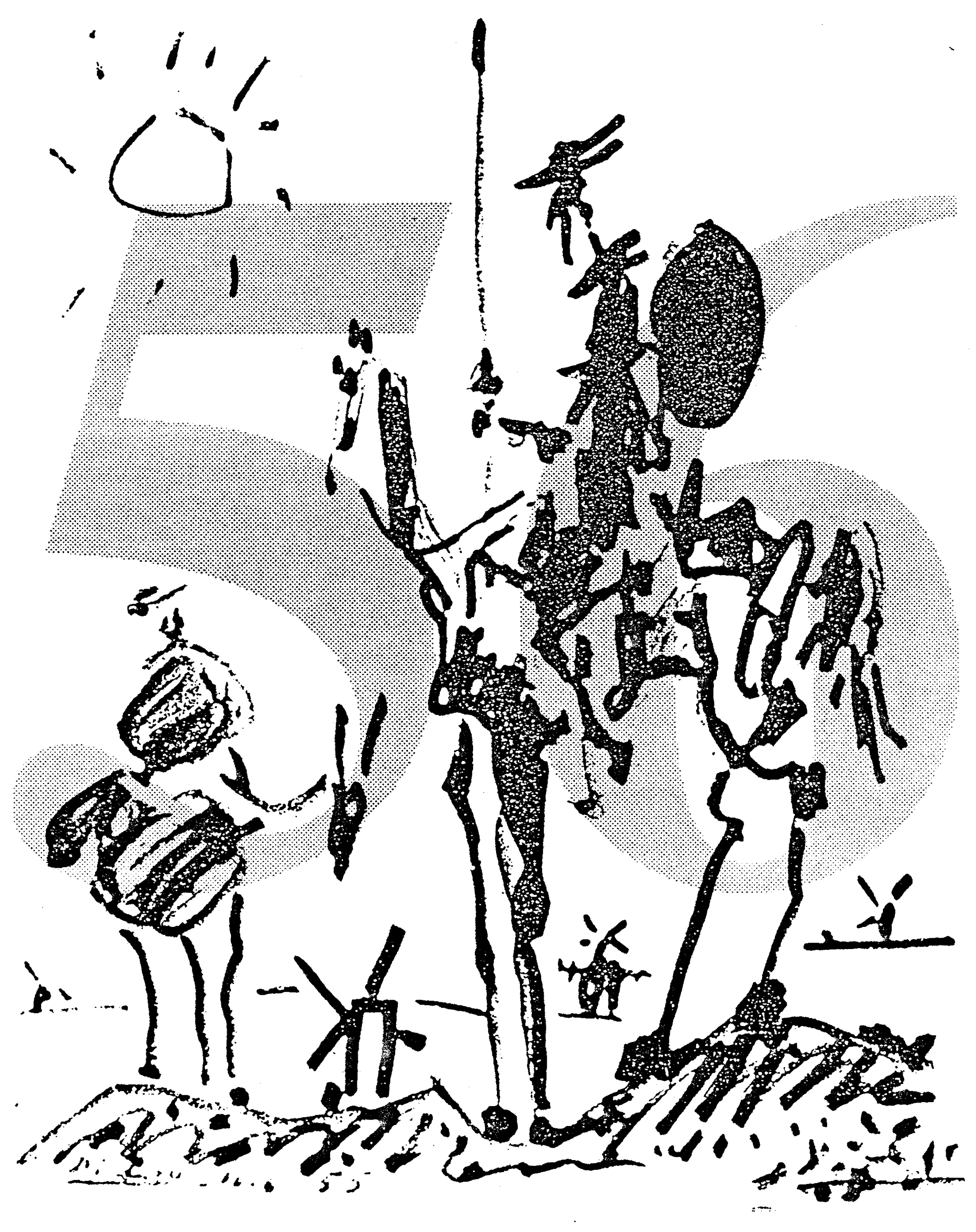
We'll also use HyperCard 2.0 to create prototype online books. Depending on the progress and direction of Jane, we may be able to use Jane as a base for prototypes at some point. We don't know who will actually create and maintain these prototypes yet.

Of course, we'll test the prototypes on real people (probably PUG members as well as Pink people) and we'll check the readings on the electrodes we'll attach to their heads.

Don Quixote

The Pink UNIX Adapter

Don Quixote
The Pink UNIX Adapter
Geoff Peck



Don Quixote and Sancho Panza
drawing by Pablo Picasso, 1955

56

Introduction

Don Quixote, the UNIX adapter for Pink, allows end users to run standard UNIX software on a Pink Macintosh while simultaneously running Pink and Blue applications. .End users will be able to exchange information between the three "worlds" in several ways — at the minimum, cut and paste and limited inter-system file access will be implemented. Since UNIX is a essentially a text-based system, Don Quixote will allow reading and writing of unformatted Pink text files from the UNIX environment. UNIX programs will be able to access Pink files, and a UNIX developer might choose to write code which can decode one or more non-text Pink file formats. Don Quixote will not perform such translations automatically. Blue text files will also be accessible to UNIX applications through a similar mechanism.

Developing Don Quixote also will help ensure that Pink contains the required facilities for third-party developers to develop other adapters. It also will serve as a substantial design and programming example of an adapter which can be distributed to third-party developers.

Detailed plans for Don Quixote are currently in process; the information contained in this section is preliminary. The description below concentrates on one of several alternatives which are currently being investigated; other alternatives are briefly described in the *alternatives* section at the end of this chapter.

Description

Don Quixote allows the end user to run unmodified binary programs which were compiled for a specific UNIX system on a specific processor, such as the AT&T System V Release 4 Application Binary Interface for Motorola 68030-class processors, or Apple A/UX Release 2.0. It may be possible to simultaneously support more than one target UNIX environment under Don Quixote.

Don Quixote will include a minimal set of standard UNIX utilities — those which are required to make the system operate in a manner conforming with basic standards. All system administration and most use of the UNIX software will be handled by Pink and the Pink finder, rather than through normal UNIX facilities.

Don Quixote is not intended to be a replacement for a standard full-featured UNIX system — rather, it is a reduced-complexity UNIX for "the rest of us" who want some or all of the capabilities of UNIX but don't want the difficulties associated with a standard UNIX .

Strategy

The strategy for Don Quixote is quite different from the path being taken by the "traditional" UNIX workstation vendors (SUN, HP, DEC, etc.), by our own A/UX group, or by NeXT. These companies are taking a complex system and putting "pretty wrappers" on top of it to make it nicer looking and easier to use. Both NeXT and A/UX are using this approach to attempt to turn a relatively traditional UNIX workstation into a personal computer. The "wrapper" approach does not address the fundamental problem — the complexity of UNIX.

Don Quixote is an <u>entirely new</u> environment, built on top of Opus/2, which provides the facilities of UNIX *without* the complexity. This is done by building a new system which conforms to relevant standard interfaces, but does not have the complex internal and administrative structure of UNIX. The benefit to the user is that Don Quixote will have many fewer "moving parts" than a traditional UNIX.

Don Quixote is an integral part of the Pink software release. It should be packaged at no additional charge with every Pink system. By virtue of the number of installed machines in the high-end Macintosh market (68020 processor or greater), the introduction of Pink with Don Quixote will immediately change the installed-base balance in the UNIX marketplace. Current UNIX software vendors will be strongly encouraged to ensure that their programs work under Don Quixote, simply due to the sheer number of Macintoshes which run Don Quixote. Don Quixote must provide a high degree of compatibility with industry standards in order to reduce the investment and time required by UNIX program developers. Availability of many UNIX packages in addition to the well-established base of Macintosh applications will immediately establish the Macintosh as the system of choice for many customers who require UNIX-based solutions.

Don Quixote, like the rest of standard Macintosh system software, is solely Apple-developed code. In certain cases, we may elect to include with Don Quixote some code which is available to the public, such as GNU software from the Free Software Foundation. If Don Quixote were to be built using existing AT&T UNIX code, Apple would have to pay a royalty to AT&T for each copy which was shipped. Since Don Quixote is completely Apple-developed, no royalties will be due, and it will be practical for us to distribute the system to the large Macintosh installed base at low cost.

Source code for Don Quixote will be readily available through the usual system software distribution channels or directly from Apple at low cost. The source code can be used as an example of how to build other adapters for Pink, as well as an excellent teaching tool in operating system courses at the college level.

The ready availability of source code will also encourage University users and advanced corporate users to use Don Quixote rather than competitors systems. These users want to or need to modify system source code to meet their special needs — to add custom network protocols, for example. Since Don Quixote consists of code written by Apple, it will not be encumbered by an AT&T UNIX license — and we can expect substantial enhancements to be written and made available to others by such advanced users. I would recommend that Don Quixote be free of any Apple licensing restrictions as well.

Don Quixote will allow the execution of virtually all user-level (as opposed to system administrative, or kernel-resident) UNIX software with little or no modification. Binary program compatibility ("ABI" or "application binary interface") will be provided between Don Quixote and specific UNIX systems, such as SunOS 4.0, A/UX, System V Release IV 68000 and/or 88000 ABI, etc. Software which does not run under one of these flavors of UNIX or which relies on features which are not supported by Don Quixote will require recompilation, and possibly source code changes, in order to run under Don Quixote. Since Don Quixote supports execution of most standard UNIX programs, however, this development will be able to be done using standard UNIX tools. This eliminates the need for UNIX developers to learn a new development environment, which has been identified as a major stumbling block to moving UNIX software to the Macintosh.

Standard UNIX utilities or their equivalent will be available for use with Don Quixote, packaged as several functional groups. A minimal set of utilities will be included with the basic distribution included with Pink. Others will be available separately through standard Apple system software distribution channels — from public distribution sources, or for a nominal charge from Apple and/or dealers. This type of packaging permits a typical office-environment Pink system to run much UNIX software with relatively little disk overhead or added complexity. Users who require additional UNIX facilities would load them when required, and would use more disk space.

Tactics

Producing a UNIX work-alike while simplifying the system is a challenging but achievable task. The first stage is identifying the appropriate standard or combination of standards with which to guide the effort — the federal information processing standard FIPS 141, IEEE's POSIX standard, AT&T's System V release 4, OSF/1, and/or X\OPEN. The standards each include system call interfaces, library interfaces, and a standard program suite. Many of the standards include optional features, which will be evaluated to see how useful they will be vs. how much complexity they add.

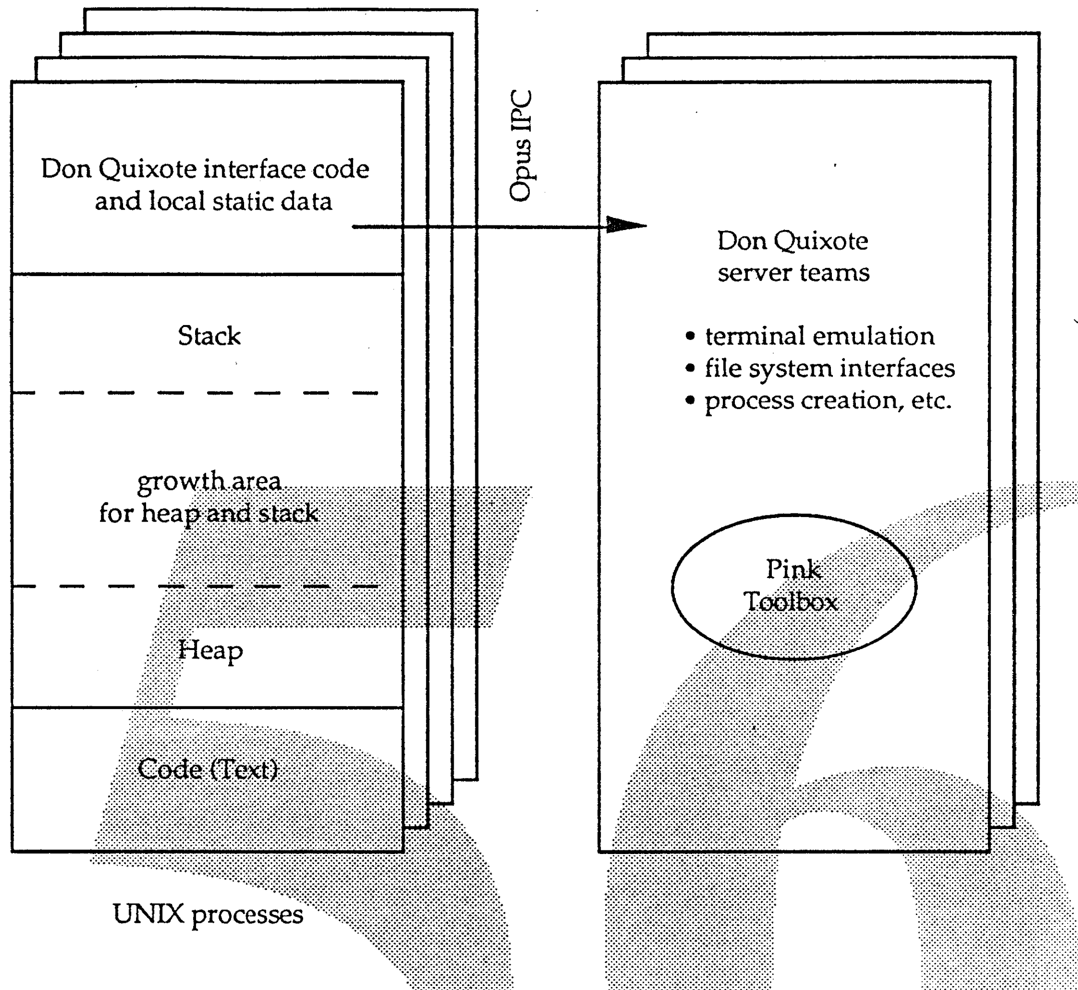
The result of the standards effort will be a single document which specifies the interfaces and programs which Don Quixote will support, and provides an initial cut at how the system will be split up into packages. After the standards document has been finalized, several implementation specifications can be developed. One implementation specification will describe how the adapter itself is structured, what internal interfaces will be required, and the impact it will have on the Pink and Opus/2 environment. Another document will list the small number of utility packages which will be developed, how these packages will be divided and installed, what the file system hierarchy will look like, and for each utility program whether it is to be developed in-house, adapted from another program, or adopted from an outside source.

A preliminary list of of the system calls which are expected to be supported by Don Quixote is included as Appendix A of this chapter.

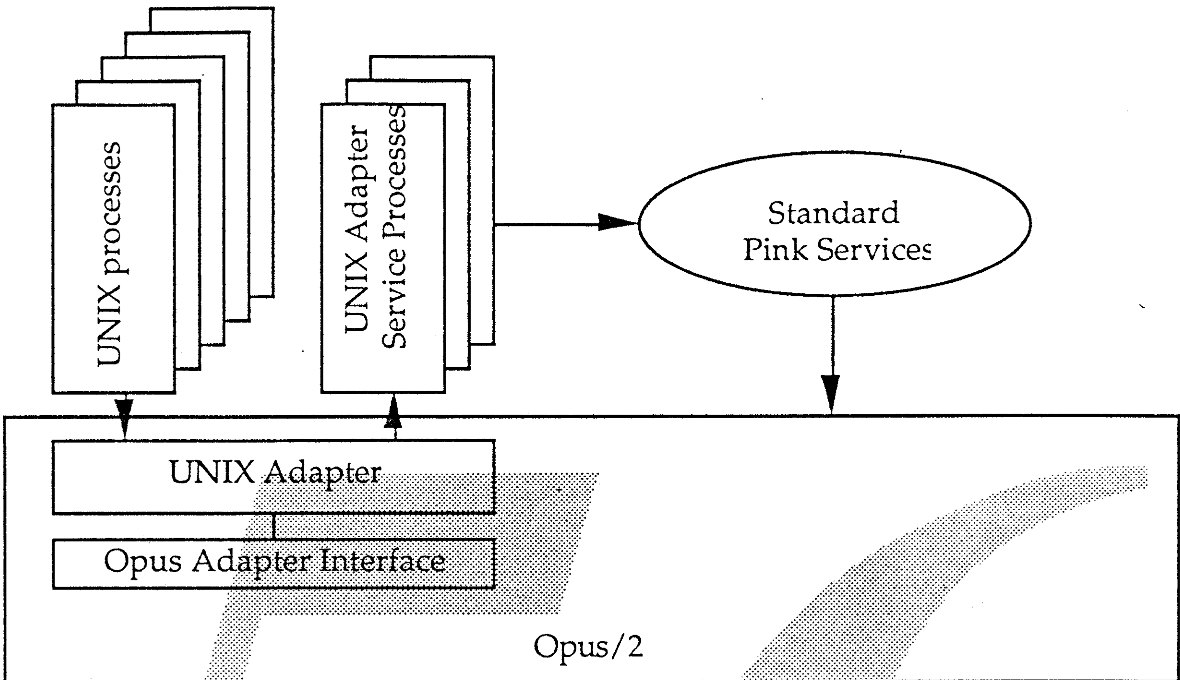
Architecture

Don Quixote is one of two adapters planned to be implemented by Apple for Pink; third-party developers may choose to implement additional adapters in the future. The Don Quixote framework will allow implementation of these adapters in a consistent manner.

Currently, two possible adapter architectures are being investigated. In the first architecture, Opus/2's exception-handling facility is used to direct UNIX process system calls and other exceptions to a small piece of code which is resident in the UNIX process address space. This code takes appropriate action for simple requests, and sends messages via Opus IPC to server processes for more complex requests. This architecture is shown below.



It is possible that this arrangement may not work due to idiosyncrasies in the way UNIX processes work, in particular with System V Release 4 dynamic linking, or that performance may not be satisfactory. An alternative architecture is to view an adapter as analogous to a device driver: a small portion of the adapter must run in privileged (kernel) mode as part of the system, and other components can run as standard Opus teams. The adapter interface allows the adapter to provides a full virtual environment for one or more address spaces, managing them in a manner different from that of Opus teams.



Currently, it is planned that Don Quixote will not implement a file system of its own — it will rely on the standard Pink file system for storage needs. If the semantics of the Pink file system differ significantly from the needs of UNIX software, a UNIX-style file system can be constructed at a later date and integrated into the system as a whole through the Pink file system switch.

Questions and Answers

In reviewing earlier versions of this document, several individuals asked a number of interesting questions.

Why is this approach better than "wrappers" around a "real" UNIX?

"Real" UNIX systems have a number of pieces which could be re-designed to run much more automatically and safely. The file system, for example, could be replaced by one which is crash-resilient, thus eliminating file-system checking (*fsck*) and the resulting delays and confusion. The UNIX-to-UNIX transfer facility (*uucp*) can be re-engineered to keep its information in a single database, rather than dozens of small control files which are scattered in several places in the file system hierarchy, and a simple administrative interface for this single database could be devised.

UNIX systems typically fail to operate in an expected manner when any one of dozens of small control files or programs are slightly incorrect or are out of place. One could envision a system which attempts to fix these files automatically, but such a system glued on top of an existing UNIX is simply adding a layer of complexity, which may or may not function properly depending on the user's particular environment.

A "real" UNIX will always, fundamentally, have quite a few pieces which may need to be adjusted by a "wizard". Even if the "wrappers" can allow a non-computer-guru user to set up a system in a standard manner, configuration changes usually are needed for a given work environment which must be performed by a "wizard". Such changes will frequently cause the "wrappers" to no longer work.

Why should Apple have two UNIX products, A/UX and Don Quixote?

Don Quixote and A/UX address two separate markets. Don Quixote provides UNIX capability for the user who occasionally wishes to run one or more UNIX programs on his or her Macintosh. A/UX provides full UNIX capability for the user who requires it, as well as the capability to run occasional Macintosh programs. Neither product meets the needs of the users of the other product.

Apple's software distribution for the Macintosh works well because little or no channel support is required. How can this apply for a complex UNIX-like system?

Don Quixote will be designed from the ground up to be easy to use and to install. The user will need to ask "do I want UNIX capabilities at this time?" If the answer is yes, the user pops in the installation diskette and runs the Installer or, if required, a souped-up installer program. Configuration and installation will be relatively automatic, and fewer files and directories will be required for most subsystems (*UUCP*, for example). The design of Opus/2 and Don Quixote will allow extensive changes in the configuration without, as is currently the case with UNIX systems, requiring re-linking of the kernel: merely dragging an appropriate facility to (or from) the system folder will have the desired effect.

What documentation would come with Don Quixote? What about supplemental documentation?

The portion of Don Quixote which is shipped with every Pink system is relatively simple and limited in capability and feature set. A single manual, no larger than the current *Macintosh Utilities Users Guide*, should suffice for the base system.

If desired, optional software components could be packaged with the appropriate manuals. Rather than including complete documentation with the software components, a supplementary manual set could be published and made available through normal book distribution channels, à la *Inside Macintosh*. Several years ago, Holt, Reinhart, and Winston published a two-volume set of paper-bound UNIX manuals which was exceptionally convenient to buy and to use.

Will Don Quixote compete with Pink for users? For developers?. What will be the impact on users?

UNIX applications and Macintosh applications are currently typically complementary, rather than competitive. Don Quixote will allow that symbiosis to continue, on a single machine. For users, then Don Quixote expands the computing opportunities, rather than forcing them to make a choice between Don Quixote and Pink.

It is possible that some developers will prefer to develop software for Apple's machines in the Don Quixote environment rather than the Pink environment. The availability of a UNIX-style development environment (both tools and programming environment) is likely to make the perceived cost of entry to application development on a Macintosh much lower. Don Quixote therefore may encourage many applications to be written for Pink-based Apple products that otherwise would not be written.

Alternatives

Here are several of the alternative implementation architectures which were considered:

Direct execution solutions

1) An Apple-developed adapter which can directly execute UNIX binaries which comply with the System V Release 4 applications binary interface (ABI).
2) An Apple-developed adapter which can directly execute UNIX binaries which are compiled specifically for this adapter.

Library-based solutions

3) An Apple-developed library which allows most UNIX source programs to be compiled and linked to execute in the Pink environment.
4) An Apple-developed library which allows some UNIX source programs to be compiled and linked to execute in the Pink environment.

Where does the code come from?

5) Port an existing UNIX kernel (possibly A/UX) to run on top of Opus/2.
6) Port and adapt existing AT&T UNIX libraries to interface to the Pink environment.

The Pink-do-nothing strategy

7) Provide a Pink adapter for A/UX, and don't provide a UNIX adapter for Pink.

The three Apple-developed alternatives which support a full UNIX interface (numbers 1, 2, and 3 above) are all roughly comparable in implementation complexity. Each requires the full set of UNIX library routines and nonprivileged system calls to be made available. These facilities must conform to established standards and must be able to pass appropriate compatibility tests. Implementing several hundred library routines and about one hundred system calls might appear on the surface to be a reasonably easy task, requiring only a few man-years of effort. However, tremendous attention to detail will be required to meet compatibility requirements. I would estimate any of these implementations to be on the order of ten to twenty man-years, with less than 10% difference in effort between the alternatives.

The fourth Apple-developed alternative (limited functionality) is of course less complex than any fully-featured system. The question is how much of a subset will be necessary to meet the needs of customers and/or developers? I believe that we can look at the market history of other such products to reach our conclusions: unless Don Quixote provides *full* functionality, we will find it extremely difficult to attract UNIX software vendors and sometimes-UNIX-users to Pink. There are a number of products which tried the library-based approach, the most notable of which were Apollo Computer's early UNIX offerings and the various adapters which allowed UNIX programs to run under DEC's VMS. None of these met with market success.

The alternatives involving AT&T code may require somewhat less development time than the fully Apple-developed alternatives; I would estimate this savings at no more than 30% of the development cost. These alternatives would require Apple to purchase an AT&T UNIX license for each system we ship with Don Quixote, or would require us to package Don Quixote as a separately licensed product. I believe that it is very important for us to offer Don Quixote to our customers on the same basis as we do other Macintosh system software — "bundled," at no additional cost to the user. Thus, Apple would have to pay AT&T for a UNIX license for each and every Macintosh which exists and is capable of running Don Quixote *at the time of product introduction*, as well as to pay AT&T a royalty for each system produced after that point. This would be a multi-million dollar expense. (Currently, $50 per system is the least expensive UNIX license; even if it could be negotiated down to $20 per system, the up-front payment to AT&T would be in excess of $20 million!)

The several library-based alternatives (3, 4, and 7) suffer from several specific technical problems:

1) UNIX programs expect that the user address space contain only information which is relevant to the program being executed, and that any system information which pertains to the program either be kept in a separate supervisory address space or at least be completely protected from access by the program. UNIX programs on occasion take (unreasonable) advantage of the degree of protection offered by this one-program, one-address-space model. At the present time, the implementation of the Pink toolbox includes a shared memory region in the address space of each process, and this shared memory region contains potentially sensitive information.
2) It will be extremely difficult to correctly emulate certain combinations of system calls without a true UNIX process model, such as *pipe*, *fork*, *exec*.
3) One of the secondary goals of Don Quixote is to be the first in a series of adapters, many of which would be developed by third parties. Add-on adapters might be particularly attractive in the Jaguar environment — due to the raw processing speed of the Jaguar processor, implementing adapters for MS/DOS, OS/2, and other operating systems which are tied to foreign instruction sets is quite feasible. Producing a UNIX compatibility library rather than a UNIX adapter will not accomplish the goal of defining the tools necessary to produce an adapter.

In light of the ineffectiveness of partial solutions, the cost of licensing AT&T code, and the technical problems with library-based alternatives, I feel that the most compelling implementations for a Pink UNIX adapter are (1) and (2) above — software which creates a full UNIX process environment within the Pink framework.

Then, the remaining question is whether to use the System V Release 4 ABI standard or to go with a standard of our own (Pink-only UNIX binaries, or A/UX binaries, for example).

Appendix A - System Call List

Following is a consolidated list of system calls obtained from the POSIX Standard (Portable Operating System Interface for Computer Environments, IEEE 1003.1) and the current 4.3BSD Berkeley UNIX system running on AppleVAX. This list needs to be reconciled with other relevant standards, particularly AT&T's System V Release 4. A similar list needs to be generated for library routines.

The third column indicates the relevant section number in the POSIX specification or, in the case of Berkeley-only system calls, "B/" followed by an indication of the class of system call. The fourth column contains working notes on the system call's origin and whether or not the call should be implemented in Don Quixote:

- P = privileged call, not implemented in Don Quixote
- C = in 4BSD compatibility library, implemented in Don Quixote
- + = added to POSIX; not in 4BSD, implemented in Don Quixote
- ? = not in POSIX, in 4BSD, may be implemented in Don Quixote
- x = not in Don Quixote

3. Process Primitives			
fork	create a new process	3.1.1	
execve	execute a file (also 5 other variants)	3.1.2	
wait	wait for process termination	3.2.1	
waitpid	wait for process termination	3.2.1	+
_exit	terminate a process	3.2.2	
exit	terminate a process	3.2.2	
kill	send signal to a process	3.3.2	
sigsetops	signal set ops (empty, fill, add, del, ismember)	3.3.3	
sigaction	examine and change signal action	3.3.4	
sigprocmask	examine and change blocked signals	3.3.5	
sigpending	examine pending signals	3.3.6	
sigsuspend	wait for a signal	3.3.7	
alarm	schedule signal after specified time	3.4.1	C
pause	suspend process execution	3.4.2	+
sleep	delay process execution	3.4.3	C
3a. 4BSD Process Primitives			
brk	change data segment size	B/Proc	
killpg	send signal to a process group	B/Proc	?
profil	execution time profile	B/Proc	
ptrace	process trace	B/Proc	
sigblock	block signals	B/Proc	
sigpause	atomically release blocked signals and wait for interrupt	B/Proc	
sigreturn	return from signal	B/Proc	
sigsetmask	set current signal mask	B/Proc	
sigstack	set and/or get signal stack context	B/Proc	
sigvec	software signal facilities	B/Proc	
vfork	spawn new process in a virtual memory efficient way	B/Proc	
4. Process Environment			
getpid	get process identification	4.1.1	
getegid	get effective group id	4.2.1	
geteuid	get effective user id	4.2.1	
getgid	get real group identity	4.2.1	
getuid	get real user identity	4.2.1	
setgid	set group id	4.2.2	
setuid	set user id	4.2.2	
getgroups	get supplementary group access list	4.2.3	
cuserid	get user name	4.2.4	+
getlogin	get user name	4.2.4	
getpgrp	get process group identification	4.3.1	
setsid	create session and set process group id	4.3.2	
setpgid	set process group id for job control	4.3.3	+
getppid	get parent process id	4.4.1	+
uname	system name	4.4.1	
time	get system time	4.5.1	C
times	get process times	4.5.2	C
getenv	environment access	4.6.1	C
ctermid	generate terminal pathname	4.7.1	+
sysconf	get configurable system variables	4.8.1	+
4a. 4BSD Process Environment			
getdtablesize	get descriptor table size	B/Proc	

getpagesize	get system page size	B/Proc	
getpriority	get/set program scheduling priority	B/Proc	x
getrlimit	control maximum system resource consumption	B/Proc	x
getrusage	get information about resource utilization	B/Proc	?
gettimeofday	get/set date and time	B/Proc	
setgroups	set group access list	B/Proc	
setpgrp	set process group	B/Proc	
setregid	set real and effective group ID	B/Proc	
setreuid	set real and effective user ID's	B/Proc	
syscall	indirect system call	B/Proc	?
5. Files and Directories			
closedir	close directory for reading	5.1.2	
opendir	open directory for reading	5.1.2	C
readdir	read entry from directory	5.1.2	C
rewinddir	position to read from start of directory	5.1.2	C
chdir	change current working directory	5.2.1	
getcwd	get working directory pathname (BSD: getwd)	5.2.2	C
open	open a file	5.3.1	
creat	create a new file or rewrite an existing one	5.3.2	
umask	set file creation mode mask	5.3.3	
link	make a hard link to a file	5.3.4	
mkdir	make a directory file	5.4.1	
mkfifo	make a fifo special file	5.4.2	+
unlink	remove directory entry	5.5.1	
rmdir	remove a directory file	5.5.2	
rename	change the name of a file	5.5.3	
stat	get file status from pathname	5.6.2	
fstat	get file status from open descriptor	5.6.2	
access	determine accessibility of file	5.6.3	
chmod	change mode of file	5.6.4	
chown	change owner and group of a file	5.6.5	
utime	set file access and modification times	5.6.6	
fpathconf	get configurable pathname variables	5.7.1	+
pathconf	get configurable pathname variables	5.7.1	+
5a. 4BSD Files and Directories			
ioctl	control device	B/File	?
chroot	change root directory	B/File	?
readlink	read value of a symbolic link	B/File	
symlink	make symbolic link to a file	B/File	
truncate	truncate a file to a specified length	B/File	
6. Input and Output Primitives			
pipe	create an interprocess communication channel	6.1.1	
dup	duplicate a descriptor	6.2.1	
dup2	duplicate a descriptor	6.2.1	
close	delete a descriptor	6.3.1	
read	read input	6.4.1	
write	write output	6.4.2	
fcntl	file control	6.5.2	
lseek	move read/write pointer	6.5.3	
6a. 4BSD Input and Output Primitives			
flock	apply or remove an advisory lock on an open file	B/IO	?
fsync	synchronize a file's in-core state with that on disk	B/IO	?
mknod	make a special file	B/IO	?
mount	mount or remove file system	B/IO	?
select	synchronous I/O multiplexing	B/IO	
sync	update super-block	B/IO	
6b. 4BSD Network and Interprocess Communication (Sockets)			
accept	accept a connection on a socket	B/Socket	
bind	bind a name to a socket	B/Socket	
connect	initiate a connection on a socket	B/Socket	
getpeername	get name of connected peer	B/Socket	
getsockname	get socket name	B/Socket	
getsockopt	get and set options on sockets	B/Socket	
listen	listen for connections on a socket	B/Socket	

recv	receive a message from a socket	B/Socket	
send	send a message from a socket	B/Socket	
shutdown	shut down part of a full-duplex connection	B/Socket	
socket	create an endpoint for communication	B/Socket	
socketpair	create a pair of connected sockets	B/Socket	
7. Device and Class-Specific Functions			
tcgetattr	get terminal state	7.2.1	
tcsetattr	set terminal state	7.2.1	
tcdrain	wait for terminal output to finish	7.2.2	
tcsendbreak	send break	7.2.2	
tcflow	suspend terminal input and/or output	7.2.2	
tcflush	discard pending terminal input/output	7.2.2	
tcgetpgrp	get foreground process group id	7.2.3	
tcsetpgrp	set foreground process group id	7.2.4	
7a. 4BSD Device and Class-Specific Functions			
vhangup	virtually ``hangup'' the current control terminal	B/Dev	?
XX. 4BSD Administrative Functions			
reboot	reboot system or halt processor	B/Admin	?
setquota	enable/disable quotas on a file system	B/Admin	P
adjtime	correct the time to allow synchronization of the system clock	B/Admin	P
swapon	add a swap device for interleaved paging/swapping	B/Admin	P
acct	turn accounting on or off	B/Admin	P
gethostid	get/set unique identifier of current host	B/Admin	?
gethostname	get/set name of current host	B/Admin	?
getitimer	get/set value of interval timer	B/Admin	?
quota	manipulate disk quotas	B/Admin	P

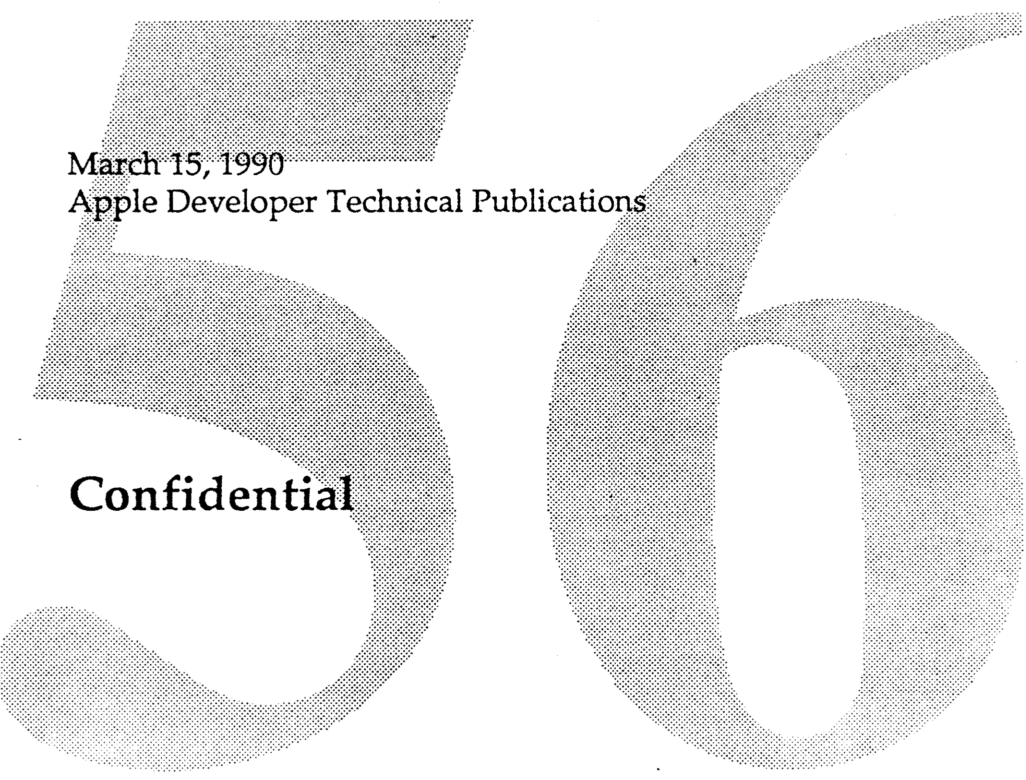
56

Technical

Documentation

56

Pink Documentation Suite Plan



March 15, 1990
Apple Developer Technical Publications

Confidential

Draft

© Apple Computer, Inc. 1990

56

Goals

The goals of the Technical Publications Pink documentation suite are to provide

- a complete description of all developer-accessible parts of the Pink system
- multiple entry points and varying treatments of the information corresponding to the skill levels and learning styles of all developers
- navigational assistance so that readers can find the information they need without becoming bewildered or wondering how to get started

The sheer volume of the information that we must provide will require us to use innovative technologies, including online delivery systems and creative ways of presenting access to the information.

Scope

In keeping with our charter, Technical Publications will produce all the documentation needed by general purpose application developers. The Networking and Communications Publications department will produce the documentation required by developers of communication software. Customer Communications will produce all the end-user documentation, including setup and utility guides and online help for end users.

Pink programming model

The Pink families of classes can be divided into two major conceptual groups: the Application Engine and the Object Toolbox. The Application Engine is an application framework consisting of families of classes that call the code the application programmer writes. Generally, every application incorporates classes from the Application Engine families. The Object Toolbox comprises the families of classes that an application programmer uses in a more conventional manner, incorporating only the classes needed to support an application's specific functions.

The following table lists some of the families that make up each group:

Application Engine	**Object Toolbox**
Application	Text
Document	Graphics
Event	Operating System Utilities
View	Operating System Services
Command	Network Services
	Sound
	Imaging
	Scripting

Pink documentation model

The documentation model strives to meet its goals while also delivering documentation when Pink ships. Since there are never enough resources and since Pink is still under development, we have designed a model that allows us to begin with the most critical and stable information, ensuring that we provide enough information to allow developers to get started. After the first customer shipments, we will continue to add information about less critical and late-developing features and to update particular components of the suite individually as required.

Our model provides

- An introduction that answers the question: "What is Pink?"
- Two different approaches to the question "How do I learn Pink?" One approach targets those who prefer hands-on learning, while the other targets those who like to learn concepts first.
- An online class and method reference which answers the question: "How do I use Pink?"
- A series of focus books tailored to specific application needs, which answer the question "How do I do awesome things with Pink?"

In addition, we will publish ancillary materials, such as human interface guidelines, a glossary, and an index.

The model is represented in the following illustration:

The Pink Documentation Suite

Introduction to Pink

The Pink Foundation

How to Write a Pink Application

Online Class and Method Reference

The Focus Library

A more complete description of the suite elements follows:

- The *Introduction to Pink* advises programmers and programming managers of the advantages of object-oriented programming in general and the Pink system in particular.
- *The Pink Foundation* describes in detail the concepts of the Pink system and the interrelationships of the families of classes. This manual, used in conjunction with the online reference, is aimed at the type of programmer who likes to learn the concepts and understand the model *before* starting to work.
- *How to Write a Pink Application* describes how to write a Pink application using the HOOPS development system. This manual is directed toward programmers who like to learn by doing. The eventual goal of this product is to be an interactive online tool.

However, if the online tool to produce this product is not available at the first customer ship of Pink, it will debut as a paper document.

- The *Online Class and Method Reference* provides descriptions of all the Pink classes and the methods that pertain to them. As our online tools become more powerful, we plan to add animations, code samples, and other interactive features to these descriptions. For plain vanilla applications, the developer will need only the online reference used in conjunction with either the tutorial or the foundation manual. The online reference will be developed first using an interim tool to search and display the information.
- The *Focus Library* consists of a series of definitive descriptions, which correspond to elements in the Application Engine and the Object Toolbox. In addition, there will be some focus books that do not correspond to such elements but which cover specific topics, such as debugging. (The current list of focus books is provided in Attachment B. We expect that additional books will be added to the library in the future as necessary.) A focus book will contain concepts interspersed with code samples. The second part of each focus will contain the appropriate descriptions from the online reference. The size of the focus manuals will be 150–400 pages depending on the topic being covered. It is expected that the N&C Pubs group will identify additional topics that they will write about to make sure that the Focus library is complete. The *Focus Library* is a new approach to documenting Apple products and will help us provide comprehensive and detailed information for those who need to know about a particular aspect of Pink without requiring those who are not interested in that detail to purchase or be overwhelmed by it. It is extensible and will be easier to revise since not all focus books will have to be updated at the same time.
- A *Human Interface Guidelines* manual similar to the one currently being designed for the Macintosh system 7.0
- An *Online Glossary* of terms
- An *Online Index* of all manuals that can be updated whenever manuals are modified or added

Every manual will include a roadmap that describes where a developer should look for various topics.

Schedules

At the present time we do not have detailed schedules. Initially, we plan to produce:

- Descriptions for many classes and methods
- A working draft of the *Pink Foundation* manual started in the form of an architectural overview document which is currently being written
- A start on the *Online Glossary*
- Detailed document design for the *Introduction to Pink*
- Detailed document design for the *Operating System Kernel* focus book (which will serve as a prototype document design for other focus books as well)

We will publish firmer plans when we have evaluated our initial efforts.

Risks and concerns

We have identified the following issues that put our plans and schedules at risk. We will be monitoring these issues during the following months.

Staffing

We have not yet determined exactly how many writers we will have, who they will be, or when they will join us. Who they are is important since the learning curve is going to be very steep. We are looking for experienced people, but since this is new technology there will still be lots to learn. We are planning to use our current writers to train new ones. While this plan will help bring the new people up quicker it will slow down the progress of the current staff.

Online tool

We have not yet identified the online tool for the *Online Class and Method Reference* or the *How to Write a Pink Application* manual. The tentative plan is to use HOOPS for online reference and Jane for interactive documents. However, this is somewhat risky since HOOPS and Jane are still under design and the full feature sets have not yet been determined. The good news is that we are collaborating with the creators to help ensure that HOOPS and Jane will meet our needs.

We plan to use the 411/Mouser format as an interim tool for the *Online Class and Method Reference.* This will allow us to get the information documented and then to port it to its eventual Pink home.

We are currently writing using unstyled text to allow us to start getting the information into the 411 format.

HOOPS availability

How to Write a Pink Application is completely dependent on HOOPS being available.

Pink progress

And, of course, we are gated by the progress of Pink itself for the completion of the *Pink Foundation*, the *Online Class and Method Reference,* the nutshells, and the *Human Interface Guidelines.*

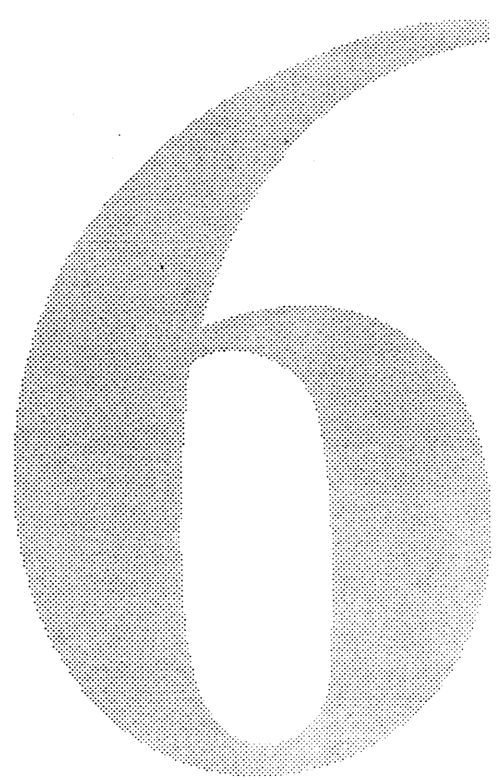
Attachment A: Summary of Products

Product	Content	Audience
Introduction to Pink	Introduction to OOP and Pink and programmers learning	Programming managers
Pink Foundation	Overview of architecture; information for people who learn, then do.	Programmers learning
How to Write a Pink Application Using HOOPS	Organized by tasks; ideal medium would be interactive and online; intended for those who learn as they go.	Programmers learning
Online Class and Method Reference	Descriptions of classes and methods	Programmers at work
Focus Library	Definitive treatment (concepts, samples, and reference); one-stop shopping for advanced topics.	Programmers at work
Human Interface Guidelines	Guidelines for application interfaces	Programmers and designers learning and at work
Online Glossary	Definitions of terms	Programmers learning and at work

Attachment B: Currently Identified Focus Library

Application Engine

Application
Document
Views
Events
Commands
Failure Handling



Object Toolbox

Text
Graphics
Operating Systems
Utilities
Network
Sound
Imaging
Scripting
Help
Building Block Views

Networking books

TBD

Other

Runtime (memory management)
File System
Debugging

56