SYSTEM SERVICES GUIDE
VOLUME 2 OF 2

BiiN™

SYSTEM SERVICES GUIDE VOLUME 2 OF 2

Order Code: 6AN9010-1XA00-0BA2

LIMITED DISTRIBUTION MANUAL

This manual is for customers who receive preliminary versions of this product. It may contain material subject to change.

BiiN™
2111 NE 25th Ave.
Hillsboro, OR 97124

REV.	REVISION HISTORY	DATE
-001	Preliminary Edition	7/88

Additional copies of this or any other BiiN™ manuals are available from:

BiiN™ Corporate Literature Dept.
2111 NE 25th Ave.
Hillsboro, OR 97124

Part VI
Program Services

This part of the *BiiN™/OS Guide* discusses program execution, concurrent programming, and scheduling.

The chapters in this part are:

Understanding Program Execution
Explains the static and dynamic structure of programs, including jobs, processes, interprocess communication, and semaphores.

Building Concurrent Programs
Shows you how to build concurrent programs, programs with multiple processes executing concurrently.

Scheduling
Explains how the system schedules processors, physical memory, and I/O devices.

Program Services contains the following services and packages:

concurrent programming service:

```
Event_Admin
Event_Mgt
Job_Admin
Job_Mgt
Job_Types
Pipe_Mgt
Process_Admin
Process_Mgt
Process_Mgt_Types
Semaphore_Mgt
Session_Admin
Session_Mgt
Session_Types
```

scheduling service:

```
SSO_Admin
SSO_Types
```

timing service:

```
Clock_Mgt
Protection_Key_Mgt
Time_Zone_Map
Timed_Requests_Mgt
Timing_Admin
Timing_Conversions
Timing_String_Conversions
Timing_Utilities
```

resource service:

```
Resource_Mgt
Resource_Mgt_AM
Resource_Types
Resource_Utilities
```

program building service:

```
Control_Types
Debug_Support
Domain_Mgt
```

```
Execution_Support
Link_By_Call
Program_Mgt
RTS_Support
```

monitor service:

```
Monitor_Defs
Monitor_Mgt
```

UNDERSTANDING PROGRAM EXECUTION 1

Contents

This chapter discusses what a program is and how it executes. It discusses the definition of a program, program structure, how a program is invoked, and how a program executes, including discussions of jobs, processes, the execution environment of processes, interprocess communication, process control, and the use of semaphores for mutual exclusion.

VI-1.1 Definition of a Program

As explained in the `Program_Mgt` package , there are four program types: executable programs, executable image modules, non-executable image modules, and views. As used in this chapter, the term *program* refers to an *executable program* or *executable image module.*

An executable program is the end product of the compiler/linker translation process. The compiler translates source code into object modules, and the linker then links the object modules into an executable program. In other words, an executable program is a program in the conventional sense of the word.

Like an executable program, an executable image module is the end product of the compiler/linker process. But unlike an executable program, it is an independently linked, protected, and potentially shareable module that provides the runtime environment of a program (for example, the language runtime system or the operating system). An executable image module contains data structures and subroutines that initialize the data structures.

Before execution, a program has a static structure; that is, it is a collection of static, passivated objects that define the elements in a program : a *program object*, a *global debug table*, an *outside environment object*, and one or more *domain objects* (which reference other objects). Sections VI-1.2 through VI-1.2.8 (Pages VI-1-2 through VI-1-8) discuss the static structure of programs.

During execution, a program has a dynamic structure; that is, it is a collection of dynamic, active objects that define the course of execution: a *job*, one or more *processes*, and one or more *stacks*. Sections VI-1.4 through VI-1.7 (Pages VI-1-9 through VI-1-17) discuss the dynamic structure of programs.

VI-1.2 Program Structure

This section discusses the static structure of programs.

A program is a network of objects rooted in a *program object*. A program object is created by the linker and referenced by a *program AD*. After creating a program, the linker passivates the objects and stores the program AD in a directory . A program consists of:

- A *program object* (Required)
- A *global debug table* (Required)
- An *outside environment object* (Required)
- One or more *domain objects* (required), each referencing:
  - A *static data object* (Required)
  - An *instruction object* (Required)
  - A *stack object* (Created at run time, referenced by a subsystem ID)

- A *public data object* (Optional)
- A *debug object* (Optional)
- A *handler object* (Required only for BiiN™ Ada programs)

Figure VI-1-1 shows the static structure of a program. (The stack object is referenced via a subsystem ID, indicated by dashed lines).

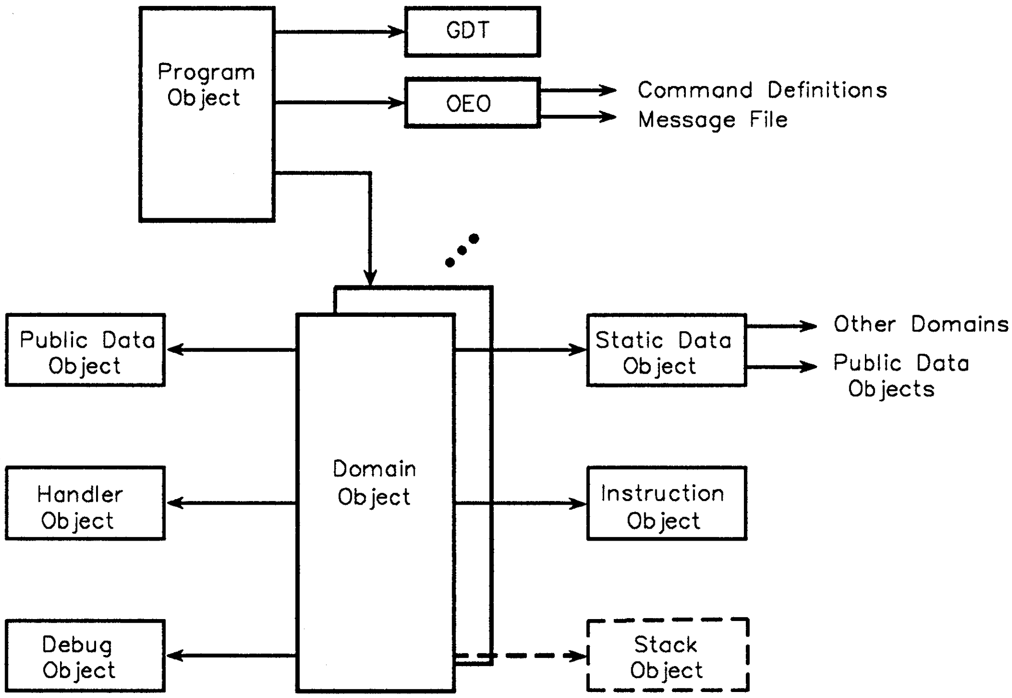

Figure VI-1-1. Static Structure of a Program

The following sections provide a brief introduction to these objects. For more detailed information, see:

- The packages `Program_Mgt`, `Domain_Mgt`, `Debug_Support`, `RTS_Support`, and `Execution_Support`.
- The *BiiN™ Systems Compiler Interface Guide.*
- The *BiiN™ Application Debugger Guide.*
- The *BiiN™ Systems Linker Guide.*

VI-1.2.1 The Program Object

The program object is created by the linker each time object modules are linked together. It serves as the root object of the program and contains:

- *The program name and version number.*
- *The main entry point of the program.* This consists of the domain AD and procedure number where execution is to begin; generally this procedure is a startup routine in the language's runtime system.

- *An AD to the Global Debug Table (GDT).* The GDT lists the compilation units that were linked to form the program. For each compilation unit, there is a reference to the debug object containing the debug information for that unit.
- *An AD to the Outside Environment Object (OEO).* The OEO references the command definitions and messages associated with the program. These are used by the command language executive (CLEX).
- *A domain AD list.* This is a list of the domains that make up the program.

Figure VI-1-2 shows the structure of a program object.

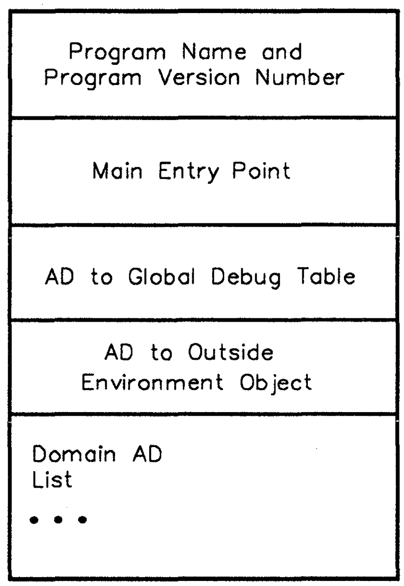

Figure VI-1-2. Program Object

VI-1.2.2 The Domain Object

Domain objects are created by the linker from object modules. Every program has one or more domains. Each domain contains:

- *An AD to a static data object.* The static data object contains ADs to external domains and public data objects so that code in this domain can call procedures and reference data in other domains. The static data object usually contains an AD to the public data object of its own domain.
- *An AD to an instruction object.* The instruction object contains the code for this domain.
- *A subsystem ID.* The ID is used to allocate and reference a stack object at runtime.
- *An AD to a public data object.* The public data object defines the data in this domain that is visible to other domains.
- *An AD to a handler object.* The handler object contains the locations of handlers that should be invoked if a fault or exception occurs.
- *An AD to a debug object.* The debug object contains information needed to debug the code in this domain.

- *A procedure table.* The procedure table lists the addresses and types of the procedures in this domain that can be called from other domains.

Figure VI-1-3 shows the structure of a domain object.

Static Data AD	0
Instruction Object AD	4
Subsystem ID	8
Not Used	12
Handler Object AD	16
Debug Object AD	20
Public Data Object AD	24
Reserved	28
Reserved	32
Reserved	36
Reserved	40
Reserved	44
Procedure Table ⋮	48

Figure VI-1-3. Domain Object

VI-1.2.3 The Static Data Object

The static data object contains data that cannot be referenced outside the current domain. If a program has only one domain, the static data object contains all variables having a global lifetime. If a program has several domains, variables referenced from another domain (for example, C `foreign` variables and Ada variables defined in packages with `pragma external`) must be allocated in the public data object.

The static data object also contains ADs to domains whose external procedures can be called from this domain, as well as ADs to objects containing data accessible from this domain.

The static data object can also contain a heap area. Heap allocation routines in the language run-time system (RTS) can resize the static data object during execution.

Figure VI-1-4 shows the structure of a static data object.

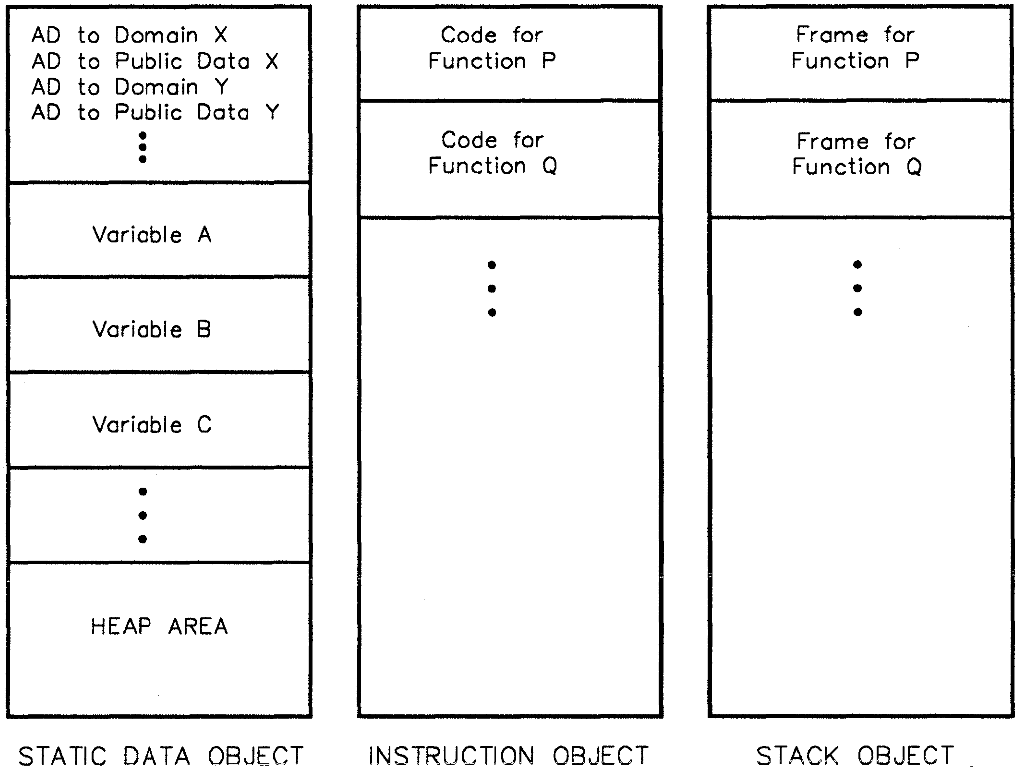

Figure VI-1-4. Static Data, Instruction, and Stack Objects

VI-1.2.4 The Instruction Object

The instruction object contains the code for all subprograms defined in this domain. It can also be used to store constant data (but not access descriptors).

Figure VI-1-4 shows the structure of an instruction object.

VI-1.2.5 The Stack Object

The stack object contains the frames used during subprogram call and return. Each frame contains the parameters, local variables, and housekeeping information related to a call.

All domains in the same subsystem and executing in the same process share a single stack object. Domains in different non-null subsystems use different stack objects.

The OS allocates the stack object when program execution begins and resizes it dynamically during execution. See Page VI-1-9 for further information.

Figure VI-1-4 shows the structure of a stack object.

VI-1.2.6 The Public Data Object

The public data object contains data that can be referenced from other domains (which have an AD to the public data object in their static data objects.)

Figure VI-1-5 shows the structure of a public data object.

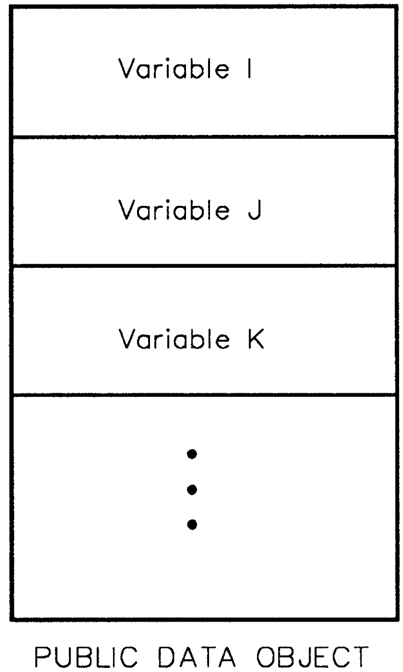

Figure VI-1-5. Public Data Object

VI-1.2.7 The Debug Object

The debug object contains compiler-generated debug information about the subprograms in the domain's instruction object.

For each subprogram, the debug object has a debug unit that contains information about the blocks, variables, constants, types, and statements in the subprogram.

Figure VI-1-6 shows the structure of a debug object.

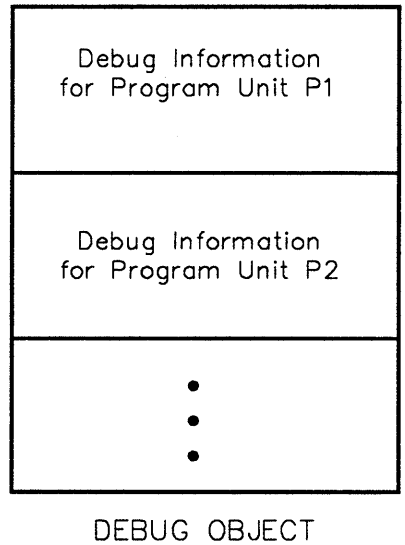

Figure VI-1-6. Debug Object

VI-1.2.8 The Handler Object

Communication between procedures typically occurs by executing explicit call/return instruction sequences. However, another mechanism is required during fault handling and exception propagation. A domain's handler object identifies the language-defined runtime system (RTS) associated with each procedure in the domain. Each RTS has a trace fault handler, a nontrace fault handler, and a number of exception handlers.

The OS handles all faults initially and handles some of them by itself. Upon encountering a fault it cannot handle, the OS needs to transfer control to the RTS fault handler corresponding to the procedure in which the fault occurred. However, the OS cannot identify the procedure's language and therefore cannot directly call the fault handler. Instead, it calls an RTS invoker routine which searches the handler object to locate the RTS's fault handler. The RTS invoker routine is defined by the linker.

When an RTS needs to propagate an exception to another subsystem, the RTS calls the OS. As with a fault, the OS then calls the RTS invoker, which searches the handler object to locate the RTS's exception handler. (If the exception needs to be propagated to another procedure in the same subsystem, the RTS, not the OS, searches the handler object to locate the exception handler.)

See the *BiiN™ Systems Compiler Interface Guide* for more detailed information about the handler object.

VI-1.3 Invoking a Program

After creating a program, the linker passivates it. Some time later, at a user's request, the BiiN™ Command Language Executive (CLEX) invokes the program in the following way:

- A user requests execution of a program by typing the program's name on a terminal.
- CLEX calls `Directory_Mgt.Retrieve` to obtain the program AD.
- CLEX uses the program's *outside environment object* (OEO) to validate the command line parameters.
- If the parameters are valid, CLEX sets up the job's environment variables and calls `Job_Mgt.Invoke_job` to create the job and its initial process.
- A CLEX-supplied initial procedure—running in the new job's initial process—calls `Program_Mgt.Run` (or `Program_Mgt.Debug`) with the program AD. `Run` (or `Debug`) then calls the program's main entry point. This activates the program, and causes the job's initial process to start executing the program's initial procedure. (This is usually a start-up routine in the language runtime system, from which control transfers to a procedure defined in one of the program's domains.)
- The program executes. After execution, control returns to CLEX (regardless of whether the program terminates normally or abnormally), and CLEX informs the user of the outcome (for example, printing any error messages).

VI-1.4 Program Execution

This section discusses the dynamic structure of programs.

A program is executed by a job. The job's initial process begins execution in one domain, obtaining instructions from the instruction object and referencing local data and procedures through the static data object.

At any time, the process may switch domains by making an interdomain call (a machine instruction) to a procedure in another domain. When this occurs, the new domain's subsystem ID is used to identify the new domain's stack object. (If the new domain is in the same subsystem as the current domain, the same stack is used). A frame is pushed on the target stack and execution continues in the new domain. A return to the original domain is accomplished by executing a return instruction using the caller's frame.

During execution, the debug object and Global Debug Table are used by the debugger to debug the program (if the debugger was invoked). Also, the handler object is used by the RTS invoker routine to identify RTS fault and exception handlers, as described earlier. (See the *BiiN™ Application Debugger Guide* and the *BiiN™ Systems Compiler Interface Guide* for more detailed information.)

During execution, a process may spawn other processes which execute concurrently. The following sections describe process behavior in greater detail.

VI-1.4.1 Sessions, Jobs, and Processes

A session is the collection of jobs executed during a user's interaction with the system. A session is usually an interactive logon/logoff period, and it typically contains several jobs.

A job represents an executing program. Each job has its own address space, memory resource, and processing resource. Scheduling, resource control, and resource reclamation are done on a per-job basis. A job can contain multiple processes executing concurrently and sharing data and resources.

A process is one thread of execution within a job. Processes share the job's resources and cooperate to perform the job's computational task. A job begins with an initial process, which can spawn other processes. See Figure VI-1-7.

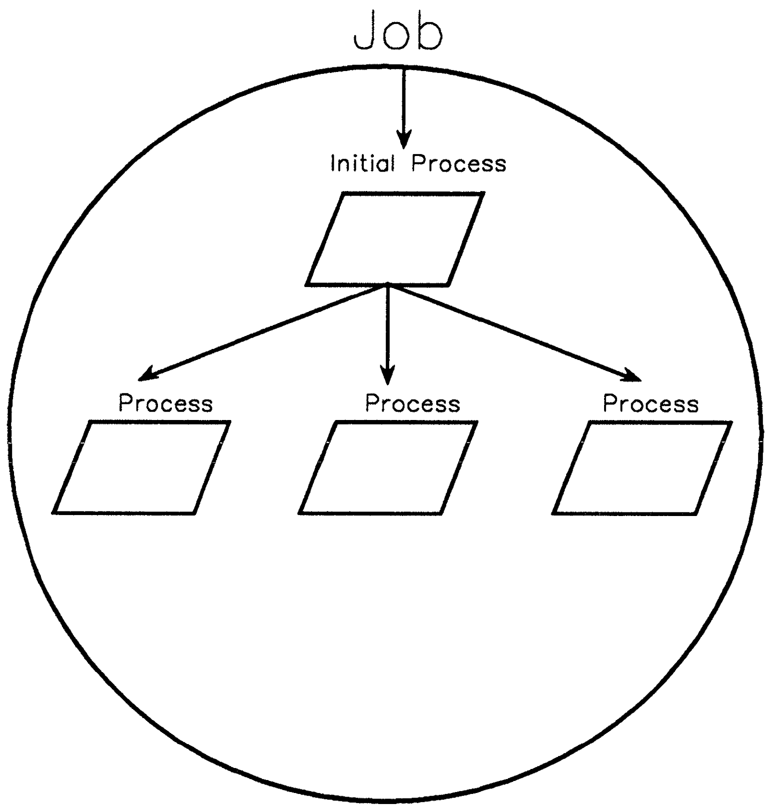

Figure VI-1-7. Job and Processes

VI-1.4.2 Process Globals

A process executes in an environment defined by its *process globals*, a list of ADs associated with the process. The entries in a process's globals are named by the `Process_Mgt_Types.process_globals_entry` enumeration type.

Most process globals entries can be modified and assigned arbitrary ADs. Your application controls the correctness of modified entries: that they are not null, have needed access rights, and reference objects of the correct type. Often your application will not need to modify the process globals entries at all; values inherited from the command interpreter or the parent process will suffice.

Table VI-1-1 describes all the process globals entries. The "Inherited?" column indicates whether an entry is inherited when a process is spawned (designated by *PS*), a job is created (designated by *JC*), or both (designated by *PS/JC*).

The "Modifiable?" column indicates whether a process globals entry can be modified. An entry can be modified when a process or job is created or by calling `Process_Mgt.Set_process_globals_entry`. In the "Modifiable?" column:

"*Admin-only*" Indicates that an entry can only be modified using the `Process_Admin` or `Job_Admin` packages.

"*Process-only*" Indicates that an entry can only be modified using `Process_Mgt` or `Process_Admin` and cannot be modified using `Job_Mgt` or `Job_Admin`.

"`Process_Admin`-*only*" Indicates that an entry can only be modified using the `Process_Admin` package.

"*Yes*" Indicates that an entry can be modified using any of the four packages (`Process_Mgt`, `Process_Admin`, `Job_Mgt` or `Job_Admin`).

"*No*" Indicates that an entry can NOT be modified using any of the four packages (`Process_Mgt`, `Process_Admin`, `Job_Mgt` or `Job_Admin`).

Table VI-1-1. Process Globals Entries

Entry	Description	Inherited?	Modifiable?
`home_dir`	Process's home directory	PS/JC	Admin-only
`current_dir`	Process's current directory	PS/JC	Yes
`authority_list`	Default authority list for objects with master ADs stored by this process	PS/JC	Yes
`id_list`	IDs for which process is granted access. First ID in list is owner ID and is default owner for objects with master ADs stored by this process. Second ID in list is group ID for BiiN™/UX processes.	PS/JC	Admin-only
`cmd_name_space`	Command name space used for retrieving command programs specified with relative pathnames	PS/JC	Yes
`standard_input`	Standard input opened device	PS/JC	Yes
`standard_output`	Standard output opened device	PS/JC	Yes
`standard_message`	Standard opened device for writing information, warning, and error messages	PS/JC	Yes
`user_dialog`	Controlling terminal. Used for operations on `/dev/tty`	PS/JC	Yes
`ux_environ`	Used for BiiN™/UX processes; null in other processes	No	**Process_Admin**-only
`lang_environ`	Used by language run-time system	PS only	**Process_Admin**-only
`site_environ`	Can be used by system administrator for site-specific purposes	No	**Process_Admin**-only
`transaction_stack`	Stack of active transactions. If the stack is not empty, the top entry is the default transaction.	No	**Process_Admin**-only
`creator`	Process that created this process, with control rights. Null if this process is a job's initial process.	No	No
`process`	AD to this process, with control rights.	No	No
`job`	Job that contains this process, with list rights and control rights.	Inherited when a process is spawned but not when a new job is invoked	No

Table VI-1-1: Process Globals Entries (cont.)			
Entry	**Description**	**Inherited?**	**Modifiable?**
`session`	Session that contains this process, with list rights and control rights.	Inherited when a process is spawned and normally when a job is invoked, but not if a job is invoked using **Job_Admin** and specifying a different session.	No, but can be implicitly modified if a job is invoked using **Job_Admin** and specifying a different session.
`name`	Optional AD to text record containing readable name for this process.	No	Process-only
`CLI_environ`	For use by Command Line Interpreter (CLEX, for example).	PS only	Process-only
`program`	For use by the OS.	PS only	Process-only
`sms`	For use by the Software Management System.	No	Process-only

VI-1.5 Interprocess Communication

This section discusses events and pipes, two basic methods of interprocess communication.

VI-1.5.1 Events

Events are a mechanism for interprocess communication with these characteristics:

- Events can be used as software interrupts, invoking *event handler* procedures and then continuing the interrupted processes.
- Events can be used to send interprocess messages. Processes can wait for events to be received. If a process is not waiting, events can be queued until the process elects to receive the events.
- Events can carry information between processes, either two words of immediate information or a pointer to a larger data structure.
- Events signalled to a job are signalled to every process in the job.
- Event clusters can be created to define additional event values or to define different process groupings:
  - An event cluster is specified by a process AD, job AD, or explicit cluster AD.
  - Each process has a predefined *local event cluster*; signalling an event using a process AD signals the local event cluster of that process.
  - A job has no cluster; signalling an event using a job AD signals the event to the local event cluster of every process in the job.
  - An explicit cluster is a *global event cluster*. Processes can associate and disassociate with global event clusters. Signalling an event using a global event cluster (AD) signals every process currently associated with the cluster.
  - The local event cluster is used for process control. See Page VI-1-16.
- Events can be signalled to remote processes or jobs.

Events are grouped in *event clusters*, each with 32 *event values*. To signal an event , you call `Event_Mgt.Signal` with an `action_record` that specifies:

`event` An event value (1 to 32).

`message` A two-word virtual address. Can be used to send immediate data or a virtual address to the data.

`destination` One of:

1. Process with control rights. Event is signaled to the process's local event cluster.
2. Job with control rights. Event is signalled to the local event clusters of all processes in the job.
3. Global event cluster with signal rights. Event is signalled to all processes associated with the cluster.

The action record specified to `Event_Mgt.Signal` is passed to any event handler or returned from any `Event_Mgt.Wait_` call that receives the event.

Each process controls how it will handle events with a particular event value by assigning the `event_status` record for that value:

`handler` Handler to establish for event. If `System.null_subprogram`, default handler (if any) is reestablished. Otherwise, handler must be in a domain with a nonnull subsystem ID.

`state` New event state. One of:

- `enabled` If the event has a handler, the handler is called for each event received. Otherwise, events are queued and can be dequeued using the `Event_Mgt.Wait_` calls.
- `disabled` Received events are discarded. If an event value's state is changed to `disabled`, any previously queued events for that value are discarded, emptying the queue.
- `handler_disabled` If the event has a handler, the handler is disabled. Received events are queued and can be dequeued using the `Event_Mgt.Wait_` calls. If the event value's state is then changed to `enabled` and the event has a handler, then the handler is called for each queued event, emptying the queue.

`interrupt_system_call` Flag indicating whether the handler can interrupt a blocked system call if the process is in the `allow_system_call_interrupt` mode. (See the `Typemgr_Support` package and `process_special_conditions .allow_system_call_interrupt` in the `Process_Mgt_Types` package for further information.)

Figure VI-1-8 shows how received events are processed.

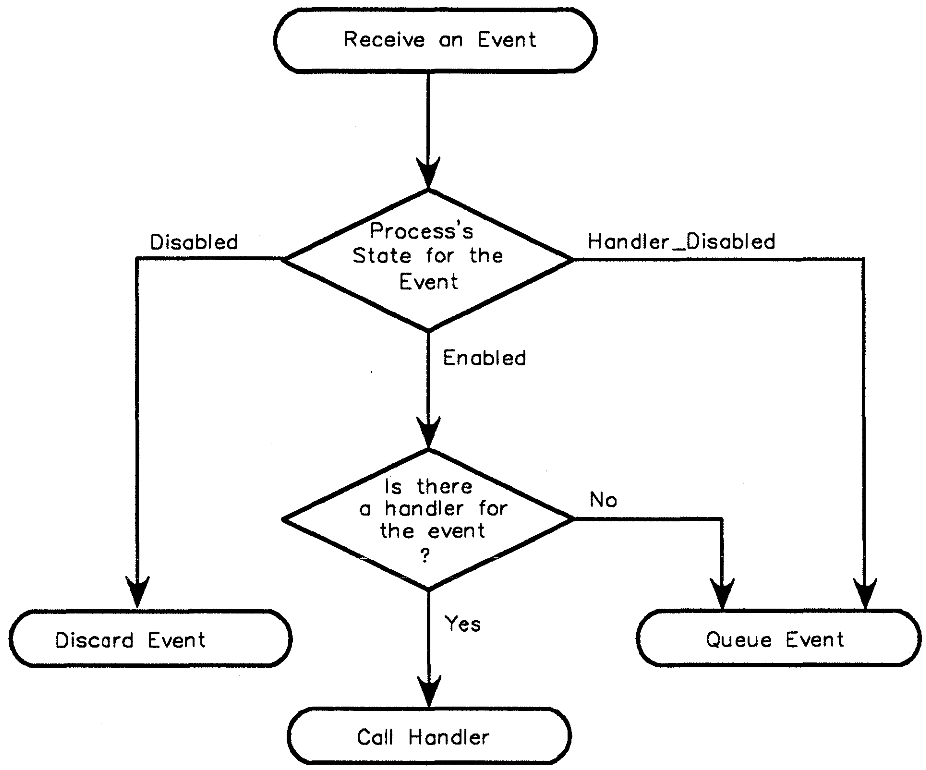

Figure VI-1-8. Events can be Handled, Queued, or Discarded.

VI-1.5.2 Pipes

A *pipe* is an object that supports one-way I/O transfers between processes.

Figure VI-1-9 shows a pipe used for interprocess communication. One process has the pipe open for output and writes data to the pipe. A second process has the pipe open for input and reads the data written by the first process. The pipe contains a fixed-size buffer used to hold data written by the first process but not yet read by the second process.

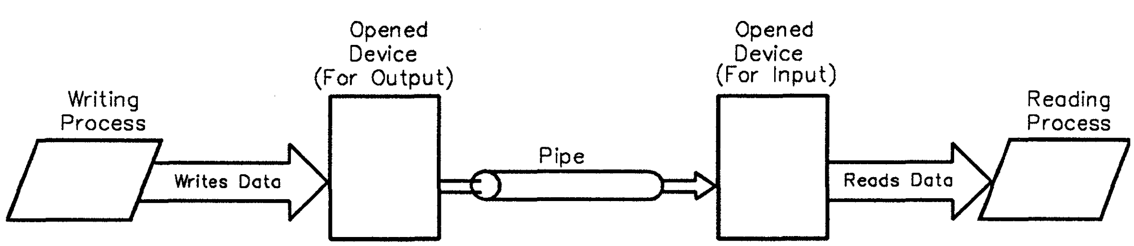

Figure VI-1-9. Pipe I/O

If a process writes to a pipe and there is not enough space in the buffer, then it can block, waiting for space to be freed by the reading process. If a process reads from a pipe but there is no data in the buffer, then it can block, waiting for data to be written by the writing process.

Pipes are one type of OS *device*. Pipes are implemented entirely in software; there are no underlying physical devices, such as terminals or disk drives, that correspond to pipes. Because pipes are software devices, they can be freely created by executing programs, limited only by the amount of virtual memory available to the process.

Pipes are useful because they eliminate the need for intermediate files by allowing the output of one program to be connected to the input of another program. This makes it easier to construct complex programs from smaller existing programs. Both the Command Language and the BiiN™/UX "shell" define an operator for piping, which takes two program invocations and connects them via a pipe. This chapter covers the procedural interface to pipes.

Pipes support the Byte Stream Access Method and the Record Access Method. These I/O access methods provide calls to open pipes for I/O, perform I/O transfers, and close opened pipes. The `Pipe_Mgt` package provides calls to create pipes, check whether pipes are open for input or output, and check whether an arbitrary object is a pipe. The `Pipe_Mgt` package description also describes the pipe implementation of the I/O access methods.

Once created, a pipe exists until no jobs reference it or until it is deallocated by calling `Pipe_Mgt.Destroy`.

VI-1.5.3 Pipes vs. Events

Both pipes and events provide distributed interprocess or interjob communication. Some comparisons will help you decide which mechanism to use for your application:

- In an application that uses pipes, a subprogram can be given an opened device and use the same code to read or write it whether the opened device is connected to a pipe, a file, or an interactive user.
- An application can send ADs and virtual addresses using events but not using pipes.
- If a message larger than two words is sent with an event, then additional message buffer space must be allocated and managed. Pipes can handle transfers of any size, even transfers larger than the pipe's buffer.
- A pipe keeps the writing process from writing too much unread data, blocking the process (or optionally raising an exception) when the pipe buffer is full. A process signalling an event never blocks and queues of pending events can grow without limit.
- Handlers can be established for both events and for pipe input (using the `Enable_input_notification` I/O access method call).

VI-1.6 Process Control

This section discusses the creation and control of processes.

VI-1.6.1 Process States

A program creates a new process within its job by calling `Process_Mgt.Spawn_process`.

Processes are controlled using local events, as described on Page VI-1-16. By sending an event to a process, you can:

- Kill it immediately
- Terminate it "gracefully", giving the process a chance to handle its own termination
- Suspend its execution until a matching `resume` event is received
- Resume its execution if it is suspended.

After a process has terminated, you can deallocate all storage used by the process by calling `Process_Mgt.Deallocate`.

Figure VI-1-10 shows major process states and the transitions between them.

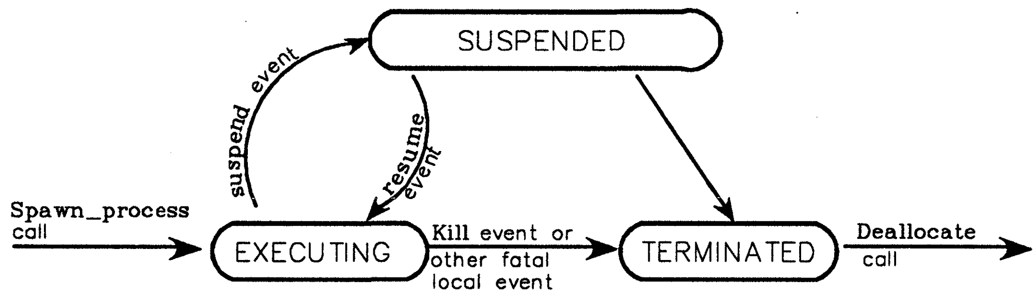

Figure VI-1-10. Major Process States

VI-1.6.2 Local Event Cluster

To kill, terminate, interrupt, suspend, or resume a process or job, signal the appropriate local event. Table VI-1-2 describes all local event values.

Table VI-1-2. Local Event Values

Value	Description	Modifiable?	Awaitable?	Default
user_1 user_2 user_3 user_4	Available for user. Not used by OS.	Yes	Yes	Enabled. No default handler.
kill	Kills process immediately, even if handling another event.	No	No	Enabled. Default handler kills process.
debug	Requests debugging. Can interrupt any other event but kill.	Event_admin- only	No, unless enabled using Event_Admin	Disabled.
termination	Requests process termination.	Yes	Yes if handler disabled.	Enabled. Default handler kills process.
interrupt	Requests abort of current operation.	Yes	Yes if handler disabled.	Enabled. Default handler kills process.
suspend	Requests suspension of process.	Yes	Yes if handler disabled.	Enabled. Default handler increments suspend/resume count. If count is now one, suspends process.
resume	Resumes process.	No	No	Enabled. Default handler decrements suspend/resume count. If count is now zero, resumes process.

Table VI-1-2: Local Event Values (cont.)				
Value	**Description**	**Modifiable?**	**Awaitable?**	**Default**
hangup	A dialup line connected to one of the process's opened devices has been hung up.	Yes	Yes if handler disabled.	Enabled. Default handler kills process.
io_complete	Available to indicate completion of an asynchronous I/O operation.	Yes	Yes	Enabled. No default handler.
local_xm	Available to signal resolution of a local transaction.	Yes	Yes	Enabled. No default handler.
gcol	Signalled each time a local GCOL run begins in the process's job.	Yes	No	Enabled. Default handler shrinks stacks if unused portions exist.
event_15 to event_32	Reserved by OS.	No	No	Disabled.

VI-1.7 Semaphores

Processes can share data. But many operations on shared data will only execute correctly if executed by one process at a time. Other processes can be excluded during such an operation by associating a *semaphore* with the shared data structure.

A semaphore is a system object that contains a count and, if the count is zero, a pointer to zero or more processes blocked at the semaphore.

The basic operations on semaphores are `P` and `V`. If a semaphore's count is greater than zero, `P` indivisibly decrements it. Otherwise, `P` blocks the calling process in the semaphore's prioritized process queue. If processes are blocked at a semaphore, `V` unblocks and dispatches the highest-priority process. Otherwise, `V` indivisibly increments the semaphore's count.

A third operation, `Conditional_P`, indivisibly decrements a semaphore's count if the count is greater than zero, returning true. If the semaphore's count is equal to zero, `Conditional_P` does nothing and returns false. A process uses `Conditional_P` to try to acquire a lock, without blocking if the lock is not available.

A semaphore can be used to lock a data structure by interpreting a *1* count to mean that the data structure is available and a *0* count to mean that the data structure is in use. Before accessing the data structure, a process calls `P`. If the data structure is available, the process continues and the semaphore's count becomes zero, indicating that the data structure is in use. If the data structure is being used by another process, the process calling `P` blocks in the semaphore's queue. After accessing the data structure, a process calls `V`. If another process is waiting, `V` dequeues the highest priority waiting process, leaving the count at zero, indicating that the data structure is still in use by the just dequeued process. If no processes are waiting, `V` increments the semaphore's count to one, indicating that the data structure is available.

A semaphore used to lock a data structure is called a *binary semaphore*. Figure VI-1-11 shows binary semaphores.

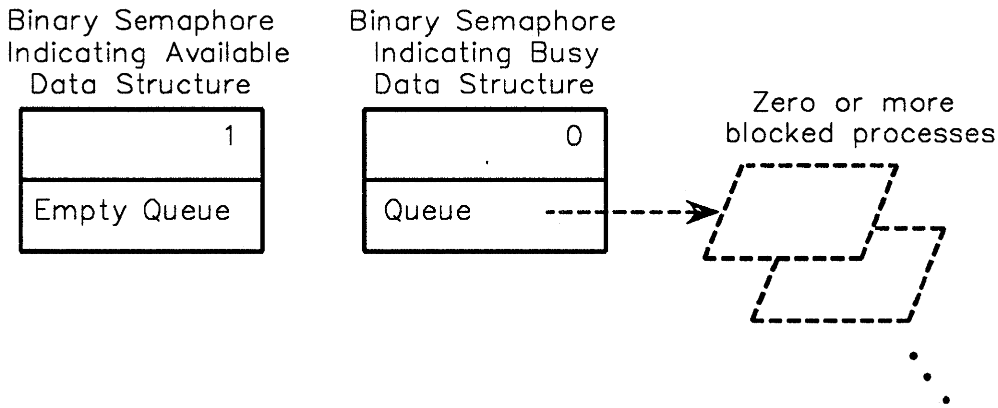

Figure VI-1-11. Binary Semaphores

A semaphore's count can also be used to count units of some resource. For example, a package that manages a buffer pool can use a semaphore's count to indicate the number of free buffers in the pool. `P` decrements the count and is called when a buffer is allocated; `V` increments the count and is called when a buffer is released. The semaphore that counts buffers can also be used to block processes that need a buffer when no buffer is available, and then to unblock a process when a buffer is released. In an implementation of the buffer pool package, a second semaphore is needed as a lock on the buffer pool data structure. A semaphore used to count units of some resource is called a *counting semaphore.*

Semaphores are supported directly by the CPU. Semaphore objects are embedded directly in their object descriptors and require no additional active memory. The `P`, `V`, and `Conditional_P` operations are implemented as single machine instructions and execute very quickly.

Semaphores are not distributed. A process can only use semaphores within its own job or within global objects on its node.

Semaphores used as locks should be held for as short a time as possible, so that other processes are blocked less often and for a shorter time. You can use the `Typemgr_Support` package to defer event handling while the process is holding a lock (only for trusted type managers).

A simple but serious bug occurs if a process uses a semaphore as a lock but never releases it for use by other processes. This could occur, for example, if the process executes a `return`, `goto`, `exit`, or `raise` statement without first calling `V`, or if an exception is propagated to the procedure in which the process is executing (preventing the process from calling `V`).

This bug causes all subsequent processes that call `P` on the lock to block indefinitely, halting all or part of an application. The section "Locking Shared Data Structures" in Chapter VI-2 shows how to write code that ensures that an acquired lock is always released.

Killing or terminating a process that uses semaphores and shared data structures can leave data structures inconsistent and leave binary semaphores with zero counts, preventing other processes from using the data structures. Because semaphores and shared data structures are normally local to a job, this problem can be avoided by killing/terminating an entire job and not just a process within a job.

If an application must acquire multiple locks before executing certain operations, then the locks should always be acquired in the same order. Consider two processes executing an application. Process A acquires semaphore C first and is blocked waiting for semaphore D. Process B acquires semaphore D first and is blocked waiting for semaphore C. Neither process can execute; each waits for resources held by the other. This is a *deadlock* or "deadly embrace" bug that can halt all or part of an application. The bug is avoided if the semaphores are always acquired in the same order, such as <C, D>.

VI-1.8 Use of Multiple Processes

This section describes three general ways to use multiple processes:

- Processes that do different tasks on data that flows from one process to the next.
- Processes that do identical tasks on different parts of a large data structure.
- Processes that have a client/server relationship in which the client sends a request to the server which sends a reply when the request has been processed.

Some operations on a stream of data can be broken into different sub-operations that can be done by different processes. The entire concurrent program resembles an assembly line where the units of work (or packets of data) flow from one worker to the next, with each doing a special part of the entire operation.

Figure VI-1-12 shows a compiler divided into separate processes to handle parsing and code generation. Data flows through a pipe between the two processes, which can access the pipe using standard I/O access methods.

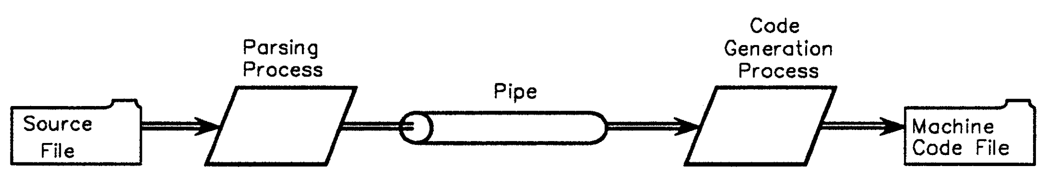

Figure VI-1-12. Processes Connected by a Pipe Speed Up a Compiler.

Some applications that can use a piped design are:

- Compilers
- Text formatters
- Format converters.

Some computations involve repeatedly doing simple transformations to large arrays of data. Figure VI-1-13 shows how such a computation can be speeded up by dividing it among multiple processes that each perform the identical calculation on a portion of the array.

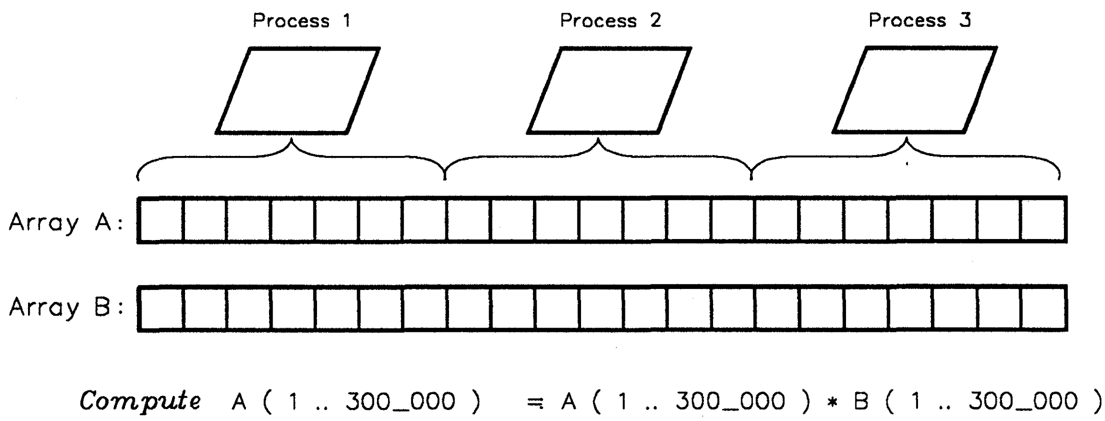

Figure VI-1-13. Multiple Processes Speed Up a Large Array Calculation.

Some applications that can use such a design are:

- Image processing
- Advanced computer graphics
- Weather models
- Models of air flow, fluid flow, heat flow, and other engineering properties
- Linear programming
- Monte Carlo simulations
- Programs that examine many possible solutions, such as a chess-playing program or programs that optimize VLSI chip designs.

Breaking an application into client and server processes can be useful when the application both requires interactive or realtime response *and* requires lengthy computations. Tasks that require lengthy processing are relegated to separate server processes. The interactive application sends requests to such server processes and can continue handling user input while the request is being processed. The server process sends a reply to its client when the request has been processed. Figure VI-1-14 shows such a design, used for a word processor with a concurrent spelling checker that checks each word entered by the user.

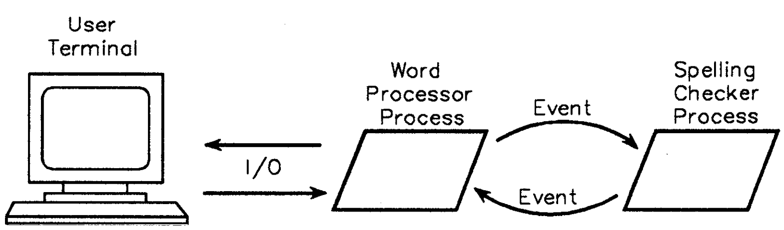

Figure VI-1-14. A Separate Spelling Checker Process Preserves Word Processor Responsiveness.

Server processes can be useful for applications such as:

- Concurrent spelling checking, grammar checking, or style checking.
- Incremental compilation of entered source code.
- Background generation of reports. For example, a process controlling a welding robot may spawn a server process that runs each hour to send operation statistics to a central computer.
- Concurrent language translation: As text is entered in one window in one language, it is translated and displayed in another window in another language. The human translator can edit either window to correct errors in text input or the computer's draft translation.

VI-1.9 Summary

- The term *program* refers to an *executable program* or *executable image module.*
- A program is a network of objects rooted in a *program object* created by the linker. It consists of a *program object*, a *global debug table*, an *outside environment object*, and one or more *domain objects*. Each domain object references a *static data object*, an *instruction object*, a *stack object* (referenced by a subsystem ID), a *public data object*, a *handler object*, and a *debug object.*
- A program is invoked by CLEX upon user request.
- A session is the collection of jobs executed during a user's interaction with the system.
- A program executes as a job. Each job has its own address space, memory resource, and processing resource. Jobs are grouped into sessions.
- A process is one thread of execution within a job. A job can contain multiple processes running concurrently and sharing data and resources.
- Each process has an execution environment defined by its process globals.
- Events provide flexible interprocess communication.
- Events are used to control processes.
- Pipes support one-way I/O transfers between processes or jobs.
- Semaphores are used to synchronize access to shared data.
- Concurrent processes can improve performance or responsiveness for a variety of applications.

BUILDING CONCURRENT PROGRAMS 2

Contents

A concurrent program is one which has multiple processes executing simultaneously within a single job. Concurrent programs are suitable for a wide range of applications and can improve program performance dramatically.

A process is one thread of execution within a job. Processes share the job's resources and cooperate to perform the job's computational task. A job begins with an initial process, which can spawn other processes. See figure VI-2-1.

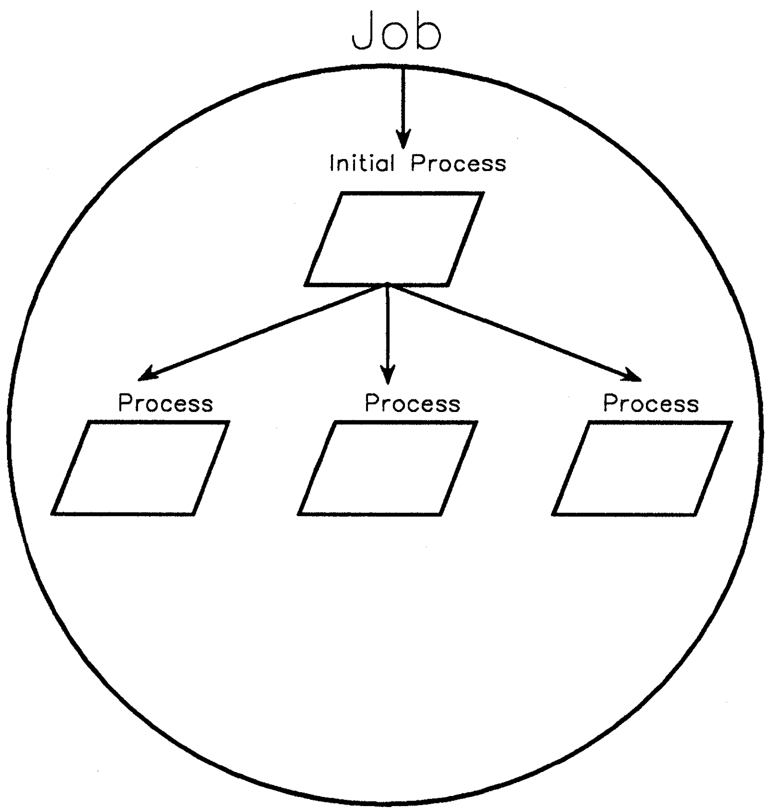

Figure VI-2-1. Job and Processes

This chapter shows you some specific techniques for building concurrent programs. You should read chapter VI-1 before this one to understand the concepts underlying programs, processes, and interprocess communication (events, pipes, and semaphores).

Packages Used:

`Event_Mgt` Manages event clusters. Event clusters provide distributed communications and software interrupts for processes.

`Pipe_Mgt` Manages pipes. A *pipe* is a one-way interprocess or interjob I/O channel. Pipes support byte stream I/O and record I/O.

`Process_Mgt` Provides public operations on processes.

`Process_Mgt_Types`
Declares types and type rights for processes.

`Semaphore_Mgt`
Manages semaphores. Semaphores can be used to synchronize concurrent access to shared data structures or resources.

This chapter shows you how to:

- Get a process globals entry
- Set a process globals entry
- Create a process
- Get process information
- Suspend and resume a process
- Terminate a process
- Signal an event
- Establish an event handler
- Wait for events
- Connect processes with a pipe
- Lock shared data structures.

Excerpts from the following examples in Appendix X-A are used:

`Compiler_Ex` Shows how a compiler can be implemented by dividing parsing and code generation between two processes connected by a pipe.

`Process_Globals_Support_Ex`
Provides calls to get and set commonly used process globals entries for the calling process.

`Symbol_Table_Ex`
Shows how a compiler's symbol table manager can synchronize concurrent access using semaphores.

`Word_Processor_Ex`
Shows how a word processor with a concurrent spelling checker can be implemented using processes and events.

Appendix X-A contains complete listings for these examples.

VI-2.1 Getting a Process Globals Entry

Calls Used:

`Process_Mgt.Get_process_globals_entry`
Gets a process globals entry.

To get a process globals entry, call `Get_process_globals_entry` with the desired entry's name. Entry names are defined by the `Process_Mgt_Types.process_globals_entry` enumeration type.

The following code is excerpted from the `Process_Globals_Support_Ex` package body:

```
    stdin:          Device_Defs.opened_device;
    stdin_untyped:  System.untyped_word;
      FOR stdin_untyped USE AT stdin'address;
  begin
    stdin_untyped := Process_Mgt.
        Get_process_globals_entry(
            Process_Mgt_Types.standard_input);
. . .
      RETURN stdin;
```

`Get_process_globals_entry` always returns a value of type `System.untyped_word`.

An optional second parameter to `Get_process_globals_entry` allows a caller to retrieve an entry from another process's globals, if the caller has control rights to the other process.

VI-2.2 Setting a Process Globals Entry

Calls Used:

`Process_Mgt.Set_process_globals_entry`
Assigns a value to a process globals entry.

To assign a process globals entry, call `Set_process_globals_entry` with the desired entry's name and its new value. Entry names are defined by the `Process_Mgt_Types.process_globals_entry` enumeration type.

The following code is excerpted from the `Process_Globals_Support_Ex` package body:

```
    opened_dev:  Device_Defs.opened_device)
. . .
    stdin_untyped:  System.untyped_word;
      FOR stdin_untyped USE AT opened_dev'address;
  begin
    if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
      RAISE Device_Defs.device_not_open;

    elsif not Access_Mgt.Permits(
        AD     =>  stdin_untyped,
        rights =>  Device_Defs.read_rights) then
      RAISE System_Exceptions.insufficient_type_rights;

    else Process_Mgt.Set_process_globals_entry(
        slot  =>  Process_Mgt_Types.standard_input,
        value =>  stdin_untyped);
    end if;
```

A value assigned to a process globals entry must have type `System.untyped_word`.

VI-2.3 Creating a Process

Calls Used:

`Process_Mgt.Spawn_process`
Creates a new process in the caller's job.

Creating a new process has two parts:

1. The program must define the initial procedure of the process in a specific way.
2. The program then creates one or more processes that execute that initial procedure.

This section's examples are excerpted from the `Compiler_Ex` package body. The first excerpt shows how a process's initial procedure is defined:

```
procedure Parse(
    param_buffer:  System.address;
      -- Address of connection record.
    param_length:  System.ordinal)
      -- Not used in this procedure, but required for
      -- process's initial procedure.
  --
  -- Logic:
  --   Do Pascal parsing using the I/O connections
  --   specified in the "conn_rec" parameter record.
is
  conn_rec:  connection_record;  -- Record containing
                                 -- parameters.
  FOR conn_rec USE AT param_buffer;
begin
. . .
end Parse;
pragma subprogram_value(Process_Mgt.Initial_proc, Parse);
```

The initial procedure must have the two parameters shown, `param_buffer` and `param_length`, whether the parameters are used or not. The `subprogram_value` pragma informs the compiler that `Parse` is an instance of the subprogram type `Process_Mgt.Initial_proc`, the type used for a process's initial procedure.

Parameters can be passed between parent and child processes by defining a record type, `connection_record` in this example, that contains the parameters as its fields. The parent process creates a connection record, fills in its fields, and passes its virtual address to the child process. The child process uses the `FOR` ... `USE AT` ... declaration to specify that its view of the connection record is at the virtual address specified by the parent.

WARNING

If a parameter buffer specified to a child process is allocated as a local variable (that is, on the stack) of the parent process, then the parent process should not terminate, or return from the call that the buffer is local to, until after the child process terminates (otherwise the buffer would be inaccessible to the child).

There are four different ways to pass information to a child process:

1. Use a parameter buffer local to the parent process. This technique is fine if the parent process does not terminate or return from the call that allocates the buffer until after the child process terminates.
2. Use a parameter buffer allocated as a separate object from the job's heap. The parent process can terminate and the buffer will continue to exist. Such a buffer can be allocated by defining an access type to whatever type is used for the buffer, and then using the Ada `new` operator to create the buffer.
3. Use a parameter buffer allocated in a package's static data area. This technique is undesirable because the buffer cannot be used by concurrent parent processes that each need to communicate with their individual children. If such a parameter buffer *is* used by concurrent parent processes, serious and hard-to-find bugs can result. If this technique is used, access to the parameter buffer should be guarded with a semaphore.
4. Communicate via changes in the child's process globals. Such changes can be specified when the child is spawned. For example, consider a child process that reads its standard input and counts lines, writing the count to its standard output. The child does not need an explicit parameter buffer; it only needs to have its standard input and standard output connected to the desired opened devices. Changes in the child's process globals can be used alone or in combination with a parameter buffer.

The second code excerpt shows how a process is created to execute a particular procedure:

```
parse_process:  Process_Mgt_Types.process_AD;
  -- Process executing "Parse".
. . .
parse_process := Process_Mgt.Spawn_process(
    init_proc    => Parse'subprogram_value,
    param_buffer => conn_rec'address,
    term_action  => (
        event =>       Event_Mgt.user_1,
        message =>     System.null_address,
        destination => this_process_untyped));
```

The initial procedure to be executed is specified using the `'subprogram_value` attribute.

The address of the parameter record is specified using the `'address` attribute.

The `term_action` parameter is optional; it indicates the action to signal when the process terminates.

VI-2.4 Getting Process Information

Calls Used:

`Process_Mgt.Get_process_state`
Gets a process's state.

`Get_process_state` produces detailed state information for a process. The process state information is contained in a record of type `Process_Mgt_Types.process_state_rec`. See the `Process_Mgt_Types` package description for more detailed information.

The state information is a snapshot and can change at the same time that the information is being retrieved. For example, `Get_process_state` may indicate that a process is executing even though it blocked while its state information was being retrieved.

VI-2.5 Suspending and Resuming a Process

Calls Used:

`Event_Mgt.Signal`
Signals an event.

`Process_Mgt.Suspend_caller`
Suspends the calling process. Is normally the last statement in a handler for the `suspend` local event.

An application can suspend a process by signaling the `Event_Mgt.suspend` local event to the process.

An application can resume a suspended process by signaling the `resume` local event to the process.

A suspend or resume event can be signalled to all processes in a job by signaling the corresponding event to the job.

Signaling either event to a process or a job requires control rights.

Each process has a ***suspend/resume*** count. A positive count is the number of suspend events received without a matching resume event. A negative count indicates the number of resume events that have been received without matching suspend events. Each suspend event received by a process increments the count, and each resume event received decrements the count. The suspend/resume count is zero when a process is created. The process is suspended whenever the count is greater than zero. Note that the resume event that matches a suspend event may be received before the suspend event.

A process can control its response to suspend events, disabling them or establishing a handler for them. A handler for suspend events can simply do whatever cleanup is needed before the process suspends itself, and then call `Process_Mgt.Suspend_caller` to suspend itself.

VI-2.6 Terminating a Process

Calls Used:

`Event_Mgt.Signal`
Signals an event.

`Process_Mgt.Terminate_caller`
Terminates the calling process.

`Process_Mgt.Deallocate`
Deallocates the storage used by a process, including the process object and process stacks.

A process can terminate itself by:

- Returning from its initial procedure
- Raising an exception that is not handled within the process
- Calling `Terminate_caller`.

A process can terminate another process or a job by signaling the `termination` or `kill` local event to the process or job. (Recall that control rights are required to signal any event to a process or job.) The difference between the two events is that processes can control their response to `termination` events but not to `kill` events.

A process may establish a handler for the `termination` event that does some cleanup and then calls `Terminate_caller`.

A process cannot modify or establish a handler for `kill` events, which terminate a process as soon as they are received; `kill` events can interrupt other event handlers.

When a process terminates, it may be desirable to free the memory that it used, by calling `Process_Mgt.Deallocate`. There is no way for a process that terminates itself to deallocate itself, so deallocation is usually handled by the parent process. If a terminated process is not deallocated, its memory can still be reclaimed by garbage collection or at job termination.

When a process creates a child process, it can specify an event to be signalled when the child terminates. The parent process can wait for that event or establish a handler for it. When the child terminates, the parent receives the termination event and deallocates the child's storage.

The following excerpt from the `Word_Processor_Ex` package body shows how the word processor signals a concurrent spelling checker process to terminate, waits for the termination event, and then deallocates the spelling checker process.

```
Event_Mgt.Signal(Event_Mgt.action_record'(
    event       => Event_Mgt.termination,
    message     => System.null_address,
      -- No message.
    destination => Conversion_Support_Ex.
                       Untyped_from_process(
                   spelling_checker_process)));
Event_Mgt.Wait_for_any(
    events => (
        child_termination_event_value => true,
        others => false),
  action => child_termination_event);
Process_Mgt.Deallocate(spelling_checker_process);
```

VI-2.7 Signaling an Event

Calls Used:

`Event_Mgt.Signal`
Signals an event.

To signal an event, call `Signal` with an *action record* that describes the event.

The `destination` and `event` fields specify which event to signal. The `message` field can be used to send a message with an event, formatted as a virtual address.

The following excerpt is from the `Word_Processor_Ex` package body. A spelling checker process has received the location of a word to check via a "word" event. If the word is misspelled, the spelling checker signals a "spelling error" event to the client process.

```
if word_mispelled then
  Event_Mgt.Signal(Event_Mgt.action_record'(
      event       => spelling_error_event_value,
      message     => (
          offset => word_event.message.offset,
          AD     => System.null_word),
      destination => word_event.message.AD));
end if;
```

The `message.offset` field of a spelling error event contains the word location, exactly as received earlier from the client process. The `message.AD` field is not used. The `destination` field is an AD to the client process being signalled. The "word" event received earlier from the client process contained this AD in its `message.AD` field.

A BiiN™ Ada representation specification can be used to pack several fields into the `message.offset` field. An excerpt from the `Word_Processor_Ex` package body illustrates this technique:

```
type word_record is record
  -- This type encodes a word location into 32 bits,
  -- allowing a word location to be transmitted
  -- using the "message.offset" field when an event
  -- is signalled.  The word processor and spelling
  -- checker are presumed to share a two-dimensional
  -- array containing the text being edited.  Words
  -- are presumed to not break across lines of the
  -- array.  A word location can thus be specified
  -- as a line number, a starting column number, and
  -- an ending column number.  The encoding limits
  -- line numbers to the range 0 .. 65_535 and
  -- column numbers to the range 0 .. 255.
  line:       System.short_ordinal;
  start_col:  System.byte_ordinal;
  end_col:    System.byte_ordinal;
end record;

FOR word_record USE
  record at mod 32;
    line      at 0 range  0 .. 15;
    start_col at 0 range 16 .. 23;
    end_col   at 0 range 24 .. 31;
  end record;
. . .
  word_event:       Event_Mgt.action_record;
    -- Receives each word to be checked.
  current_word:     word_record;
  FOR current_word USE AT word_event.
      message.offset'address;
    -- Overlay used to extract word location.,
```

VI-2.8 Establishing an Event Handler

Calls Used:

`Event_Mgt.Establish_event_handler`
Assigns handler and state for an event. Returns previous handler and state.

Establishing an event handler has two parts:

1. The program must define the handler procedure in a specific way.
2. The program must call `Establish_event_handler` to connect the handler to the event.

This section's examples are excerpted from the `Word_Processor_Ex` package body. The first excerpt shows how a handler procedure is defined:

```
procedure Spelling_error_handler(
    action:  Event_Mgt.action_record)
  --
  -- Operation:
  --   Handler invoked for each 'spelling error'
  --   event.
is
  misspelled_word:  word_record;
  FOR misspelled_word
      USE AT action.message.offset'address;
    -- Overlay used to extract word location.
begin
  -- Code to handle misspelled word goes here.  For
  -- example, this code could highlight the
  -- misspelled word on the display and ring the
  -- terminal's bell.

  null;
end Spelling_error_handler;
  pragma subprogram_value(
      Event_Mgt.Event_handler,
      Spelling_error_handler);
```

A handler procedure must have the `action` parameter shown, which is the event that invokes the handler. The `subprogram_value` pragma informs the compiler that `Spelling_error_handler` is an instance of the subprogram type `Event_Mgt.Event_handler`, the type used for all event handlers.

The second excerpt shows how the word processor process establishes this handler:

```
  old_event_status:  Event_Mgt.event_status;
    -- Saves previous event status for the
    -- spelling_error local event, so the previous
    -- status can be restored before exit.
. . .
  old_event_status := Event_Mgt.
      Establish_event_handler(
          event  => spelling_error_event_value,
          status => (
              handler =>
                 Spelling_error_handler'
                     subprogram_value,
                 state   => Event_Mgt.enabled,
                 interrupt_system_call => false));
```

When a subprogram establishes an event handler, and the subprogram is not the initial procedure or final procedure for its process, then it is good manners for the subprogram to restore the previous event status before returning to its caller:

```
  old_event_status := Event_Mgt.
      Establish_event_handler(
          event  => spelling_error_event_value,
          status => old_event_status);
    -- Reestablish previous event status.
    -- Value returned is never used.
```

VI-2.9 Waiting for Events

Calls Used:

`Event_Mgt.Wait_for_all`
Wait for all of a set of events within a cluster.

`Event_Mgt.Wait_for_any`
Wait for any of a set of events within a cluster.

`Wait_for_any` is used to wait until any of a set of events within a cluster is received. The first event in the set that is received is assigned to an action record output parameter. The following excerpt from the `Word_Processor_Ex` package body shows the spelling checker process waiting for a word to be checked.

```
      word_event:      Event_Mgt.action_record;
        -- Receives each word to be checked.
      current_word:    word_record;
      FOR current_word USE AT word_event.
          message.offset'address;
        -- Overlay used to extract word location.,
  . . .
        Event_Mgt.Wait_for_any(
            events => (word_event_value => true,
               others => false),
            action => word_event);
```

`Wait_for_all` is used to wait until all of a set of events within a cluster have been received. The received events are assigned to an array of action records. The following excerpt from the `Compiler_Ex` package body shows a parent process waiting for two child processes to terminate.

```
    term_events:  Event_Mgt.action_record_list(2);
      -- Array that receives termination events of the
      -- two child processes.
  . . .
    Event_Mgt.Wait_for_all(
        events =>
            (Event_Mgt.user_1 .. Event_Mgt.user_2 =>
                 true,
             others => false),
        action_list => term_events);
```

VI-2.10 Connecting Processes with a Pipe

Calls Used:

`Pipe_Mgt.Create_pipe`
Creates a pipe.

`Byte_Stream_AM.Ops.Open`
Opens a device.

The following excerpt from the `Compiler_Ex` package body shows how a pipe is created and opened.

```
compiler_pipe:  Pipe_Mgt.pipe_AD;
  -- Pipe that connects "Parse" and "Code_gen"
  -- processes.
. . .
compiler_pipe := Pipe_Mgt.Create_pipe;

conn_rec := (
    source_code  => source_code,
    machine_code => machine_code,
    listing      => listing,
    parse_out    => Byte_Stream_AM.Ops.Open(
        Pipe_Mgt.Convert_pipe_to_device(
            compiler_pipe),
        Device_Defs.output),
    code_gen_in    => Byte_Stream_AM.Ops.Open(
        Pipe_Mgt.Convert_pipe_to_device(
            compiler_pipe),
        Device_Defs.input));
```

The opened device ADs for the two open ends are stored in a "connection record" that is passed by address to each child process. Each child process can read the connection record and use the opened devices in it.

The `Parse` process writes the results of its parsing to the `conn_rec.parse_out` opened device, the output end of the pipe. The `Code_gen` process reads the same parse results from the `conn_rec.code_gen_in` opened device, the input end of the pipe.

VI-2.11 Locking Shared Data Structures

Calls Used:

`Semaphore_Mgt.Create_semaphore`
: Creates a semaphore.

`Semaphore_Mgt.P`
: Enters/locks/waits at a semaphore. If the semaphore's current count is greater than zero, indivisibly decrements it. Otherwise, blocks the caller in the semaphore's prioritized process queue.

`Semaphore_Mgt.V`
: Unlocks/leaves/signals a semaphore. If processes are blocked at the semaphore, unblocks and dispatches the highest-priority process. Otherwise, indivisibly increments the semaphore's current count.

A data structure shared by multiple processes can be locked by locking an associated semaphore. To ensure that all processes observe the locking protocol, the data structure can be managed by a BiiN™ Ada package that handles all access to it. The `Symbol_Table_Ex` package manages a symbol table using such a locking protocol.

The package body creates the symbol table at package initialization; the associated semaphore is created in the same code block:

```
    lock:  Semaphore_Mgt.semaphore_AD;
      -- Used to lock symbol table while a process
      -- is accessing it.
. . .
-- PACKAGE INITIALIZATION
begin
. . .
  symbol_table.lock := Semaphore_Mgt.
      Create_semaphore;
    -- Lock initially indicates table is available.
    -- First "P" on lock will succeed.
```

Each operation provided by the `Symbol_Table_Ex` package locks the semaphore at the beginning of the operation and unlocks the semaphore on all return and exception paths. The following excerpt is from the `Read_symbol_data` implementation in the package body. Note that the semaphore is locked once, but unlocked at each of several different exit paths.

```
begin

  Semaphore_Mgt.P(symbol_table.lock);
. . .
    for i in 1 .. symbol_table.length loop
      if symbol_table.value(i).name =
          fixed_width_name then
        Semaphore_Mgt.V(symbol_table.lock);
        RETURN symbol_table.value(i).data;

      end if;
    end loop;
      RAISE no_such_symbol;

  end if;

  -- This call to "V" is never reached in the
  -- current implementation.  The call is included
  -- as a safeguard in case code changes make it
  -- reachable.
  Semaphore_Mgt.V(symbol_table.lock);

  exception
    when others =>
      Semaphore_Mgt.V(symbol_table.lock);
      RAISE;  -- Reraise exception
              -- that entered handler.

end Read_symbol_data;
```

SCHEDULING 3

Contents

This chapter explains how jobs and processes are scheduled. It discusses the scheduler's objectives and tasks, scheduling service objects (SSOs), CPU scheduling, memory scheduling, and I/O scheduling.

VI-3.1 What the Scheduler Is

The scheduler is a collection of hardware and software entities whose purpose is to schedule the execution of jobs (and thus processes).

The scheduler is designed for multi-user systems, provides support for real-time applications, and withholds explicit control of scheduling from the user.

The scheduler is not intended to be replaceable; instead, the system administrator can tailor a job's scheduling parameters to suit specific requirements.

VI-3.2 The Scheduler's Objectives

The scheduler's general objective is efficient use of the system's resources. Specifically, it seeks to:

- Maximize resource utilization
- Maximize system throughput
- Minimize response time for interactive users
- Avoid starvation of jobs
- Degrade gracefully under load
- Minimize thrashing.

To accomplish these objectives, the scheduler is designed to favor:

- Interactive jobs
- I/O-bound jobs
- Jobs with small working sets
- Short jobs.

and to handicap:

- Noninteractive jobs
- CPU-bound jobs
- Jobs with large working sets.

VI-3.3 The Scheduler's Task

A job needs three resources to execute: physical memory, processor time, and I/O devices. The scheduler attempts to balance the job's need for these resources against their availability and maximize resource utilization for all jobs in the system.

Thus, the scheduler's task is threefold: CPU scheduling, memory scheduling, and I/O scheduling. These are discussed in the following sections.

VI-3.4 CPU Scheduling

This section discusses CPU scheduling.

VI-3.4.1 CPU Scheduling Model

When a job is invoked (see Chapter VI-1), it is enqueued on a *scheduling port* served by a *scheduling daemon*. Thereafter, scheduling occurs at three different levels:

- *High-level scheduling* schedules jobs.
- *Medium-level scheduling* assigns priorities to processes.
- *Low-level scheduling* dispatches processes for execution on a processor.

VI-3.4.1.1 High Level Scheduling

When the scheduling daemon is activated, it removes a job from the scheduling port and schedules it by enqueueing the job's initial process at the end of one of the queues in a *dispatching port*. The port has 32 queues, ordered in priority from 0 (lowest) to 31 (highest). (Note: Priorities 16-31 are reserved by the OS and never used by user processes.) A process enqueued in this manner is said to be *in the mix*. Putting a process in the mix is called *high-level scheduling*. See Figure VI-3-1.

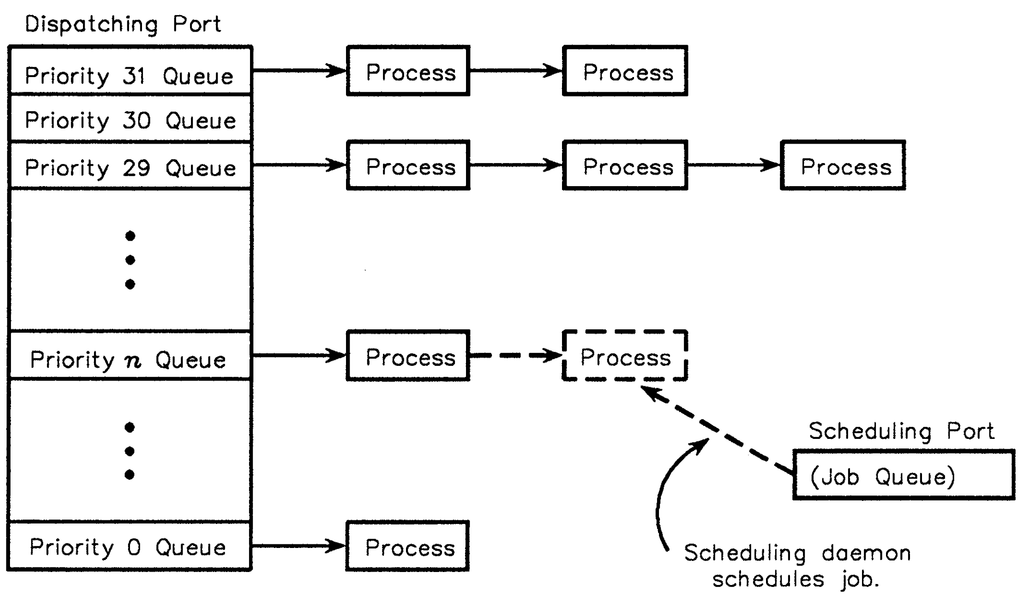

Figure VI-3-1. High-level Scheduling

VI-3.4.1.2 Low Level Scheduling

Each processor has a pointer to the dispatching port. When a processor is available to execute a process, it dequeues the first process from the highest numbered, non-empty queue in the port, and executes it. This is called *low-level scheduling* or *dispatching*; it is done by microcode, not software. See Figure VI-3-2.

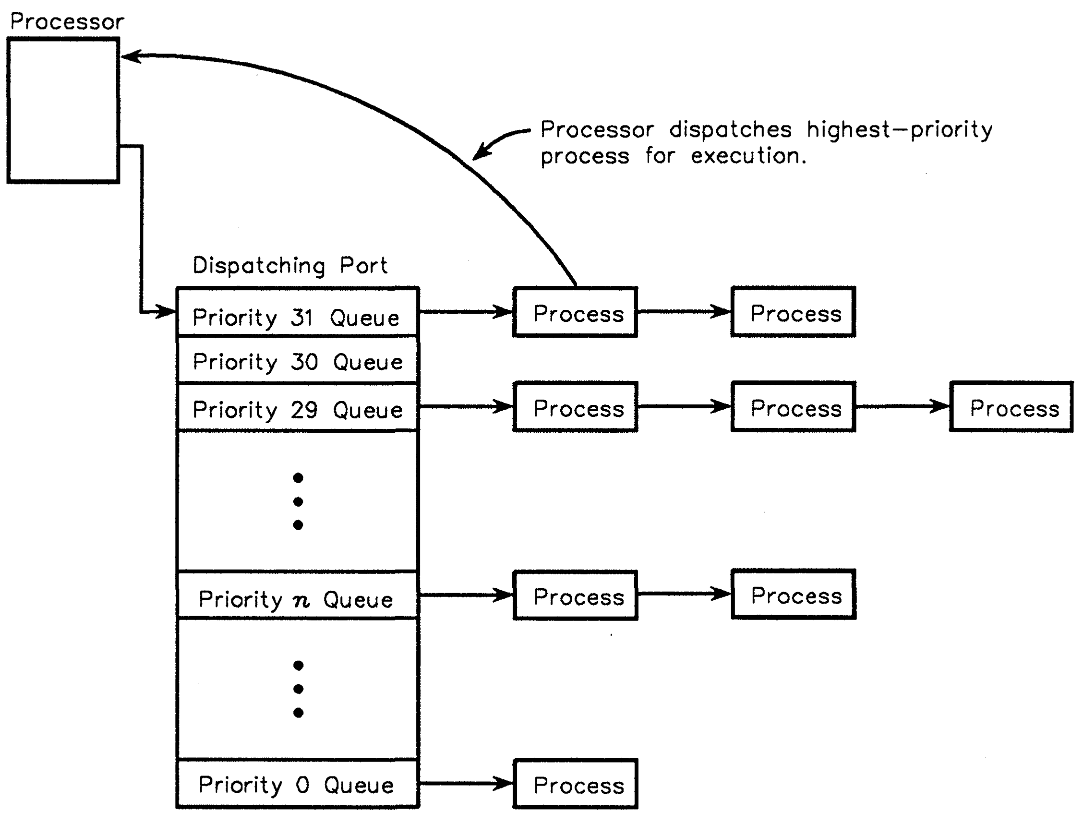

Figure VI-3-2. Low-level Scheduling

VI-3.4.1.3 Processor Preemption

It is possible for a running process to be preempted (forced to relinquish the processor) by a process waiting in the dispatching port. Whether this occurs depends on the processes' relative priorities and the system's *preemptive threshold.* Currently the threshold is *8*: if an interrupt handler or a process with a priority greater than or equal to *8* is ready to run, it will preempt a handler or process running with a lower priority.

Note that the preemptive threshold may change.

See Pages VI-3-6 and VI-3-7 for further information about process priorities.

VI-3.4.1.4 Classes and Priorities

Each job has a *scheduling service object (SSO)* that determines the type of scheduling service the job receives. Among other things, the SSO defines the job's *service class* and *priority.*

There are four service classes: *real-time*, *time-critical*, *interactive*, and *batch*. All the processes in a job have the job's service class; a job's service class never changes.

There are 32 priorities, corresponding to the priorities in the dispatching port.

See Page VI-3-6 for further information about service classes, priorities, and SSOs .

VI-3.4.1.5 Processor Claim and Job Time Limit

Each job has a *processor claim* that defines the *number* of time slices available to the job's processes in a scheduling cycle and a *time limit* that defines the total *processing time* available to the job (and its descendant jobs).

All jobs have the same processor claim, but the *length* of the time slice given to a process is determined by the process's priority.

A job's time limit is determined by by the `time_limit` parameter in the `Job_Mgt.Invoke_job` function. The exact interpretation of `time_limit` is subtle; see `Invoke_job` for further information.

When a time slice occurs, a *time-slice fault-handler* checks the processor claim:

- If it is nonnegative, the time-slice fault-handler reduces it by one and gives the process another time slice by putting it at the tail of its priority queue in the dispatching port.
- If it is negative, the time-slice fault-handler triggers a *resource-exhaustion fault-handler*, which checks the job's time limit. If the limit has been exceeded, the job is terminated; if not, the resource-exhaustion fault-handler replenishes the processor claim (charging it against the job's *Resource Control Object (RCO)*), and continues job execution.

VI-3.4.1.6 Medium Level Scheduling

The scheduling daemon puts real-time, time-critical, and interactive jobs into the mix immediately, but puts batch jobs in a waiting queue until system load allows them to be put in the mix. Once a process is in the mix, its scheduling depends on its priority, service class, and dynamic behavior. This is called *medium-level scheduling*, and is performed by hardware and the time-slice fault-handler. The following summarizes medium-level scheduling after a job has been put in the mix:

- **Real-time processes:**
  - A real-time process is not subject to *time slice* faults; that is, it executes until it terminates or blocks for I/O.
  - If it blocks for I/O, hardware returns it to the front of its priority queue in the dispatching port when the I/O completes.
  - It is up to the software designer to ensure that a real-time process does not starve other real-time processes and keep them from executing for too long a period.
- **Time-critical processes:**
  - A time-critical process is subject to *time slice* faults. When a time slice fault occurs, it is handled as described in Section VI-3.4.1.5 on Page VI-3-5.
  - If a time-critical process blocks for I/O, it is treated like a real-time process.
- **Interactive and batch processes:**

- An interactive or batch process is subject to *time slice* faults like a time-critical process and is treated in the same way, with one exception: if it receives an additional time slice, the time-slice fault-handler *lowers* the process's priority and places it at the tail of its new (lower) priority queue in the dispatching port.
- If an interactive or batch process blocks for I/O, the time-slice fault-handler *raises* the process's priority to the priority of the requested I/O device, and places it at the tail of its new (higher) priority queue in the dispatching port when the I/O completes. This allows the process to issue several I/O requests for the device at the higher priority.
- Note that the scheduling discipline for real-time and time-critical jobs is based on fixed priorities, but the scheduling discipline for interactive and batch jobs is based on dynamic, resource-driven priorities. See Page VI-3-7 for further information.

VI-3.4.2 Scheduling Service Objects (SSOs)

A *Scheduling Service Object* (SSO) is associated with a job when the job is invoked. The SSO determines the type of scheduling the job receives.

The system administrator is responsible for creating different types of SSOs and controlling access to them, thus controlling the type of service granted to different jobs (see the `SSO_Admin` package).

The SSO determines the job's service class, SSO priority, time slice, memory type, initial age, and age factor.

VI-3.4.2.1 Service Classes

Service class denotes the general class of service a job is to receive. Four service classes are defined: realtime, time-critical, interactive, and batch.

Real-time jobs are executed in real time. They have very high priority and an infinite time limit. They run in frozen memory, and are not subject to the scheduling process. They are preemptive (given the current preemptive threshold) and always in the mix. If they block for I/O, the hardware reschedules them as soon as the I/O completes.

Time-critical jobs have less stringent time constraints than real-time jobs. They have the same priority as real-time jobs, but a finite time limit (when a time slice expires, they are rescheduled or terminated). They need not run in frozen memory, since their time constraints can tolerate page faults. Like real-time jobs, they are preemptive (given the current preemptive threshold) and always in the mix.

Interactive jobs involve interaction between a user and a job (an editing session, for example). Interactive jobs run in normal memory, have a finite time limit, and have a lower priority than real-time and time-critical jobs.

Batch jobs are background jobs with no attached user. Like interactive jobs, they run in normal memory, have a finite time limit, and have a lower priority than real-time and time-critical jobs.

VI-3.4.2.2 SSO Priority

SSO Priority is the job's SSO priority. SSO priorities are defined as follows (higher values indicate higher priority):

16 - 31	Reserved for interrupt handlers; not available for program execution.
15	Timing daemon.
12 - 14	Real-time and time-critical jobs.
11	Scheduler and other well-behaved system jobs.
0 - 10	Interactive and batch jobs.

As noted earlier, a handler or process with a priority greater than or equal to the *preemptive threshold* will preempt a processor from a handler or process running at a lower priority. A handler or process with a priority lower than the preemptive threshold cannot preempt a processor. The current preemptive threshold is *8*; it may change in the future.

VI-3.4.2.3 Time Slice

Time slice is the amount of processing time assigned to each process in the job in each dispatching cycle. (It does not include time spent on such incidents as interrupts, processor preemption, or waiting at a port or semaphore).

When a process exhausts its time slice, it is handled as described in Section VI-3.4.1.5 on Page VI-3-5.

For additional information about how time slices are interpreted for different classes of jobs, see `time_slice_enabled`, `time_slice_reschedule`, and `time_slice` in `SSO_Types.SSO_Object`.

VI-3.4.2.4 Memory Type

Memory type is the type of memory in which the associated job should run. There are two types of memory: *frozen* and *normal*. Frozen memory is nonswappable, nonrelocatable memory; it is used for jobs that cannot tolerate page faults (real-time jobs, for example). Normal memory is swappable and relocatable.

VI-3.4.2.5 Initial Age

Initial age is a job's age when it first enters the scheduler's waiting queue of swapped-out jobs (see page VI-3-9). Larger values indicate *older* jobs. The job at the head of the queue is the oldest job and will be scheduled next. Giving a job a large initial age helps move it to the head of the queue more rapidly.

VI-3.4.2.6 Age Factor

Age factor is the rate at which a job ages in the scheduler's waiting queue. On every scan of the waiting queue, the age factor is added to the job's age to determine a new age. The larger the aging factor, the faster a job ages, and the sooner it rises to the front of the waiting queue.

Note that care should be used before assigning an age factor of 0 to a job. Such a job will never age, and may therefore starve in a busy system.

VI-3.4.3 Resource-Driven Priorities

A single, fixed priority (SSO priority) is used to schedule real-time and time-critical jobs, and their priority is unaffected by resource usage. In contrast, scheduling for interactive and batch jobs uses several priorities and is dynamically driven by resource usage.

VI-3.4.3.1 Priorities Used

The priorities used in scheduling interactive and batch jobs are:

SSO priority — The priority defined in the job's SSO.

Base priority — The lowest priority a process can have.

A process's base priority is set when the process is created. The base priority of an initial process in a job is the job's SSO priority. The base priority of a spawned process is the base priority of its parent process.

The System Administrator can change a process's base priority to any value; a user can change it to a value less than or equal to the job's SSO priority.

Changing a job's base priority is accomplished by changing the base priorities of all the job's processes.

Resource priority — The priority assigned to a particular resource.

When a process blocks on a resource, its priority is raised to the resource priority (unless its priority is already higher, in which case its priority remains unchanged).

After using a resource, a process must return to its base priority. Each resource class specifies the amount of time in which this must occur. The process's priority is decreased linearly from the resource priority to the base priority in the specified amount of time.

Running priority — The priority at which an interactive or batch process is currently running.

Running priority is determined by the other priorities.

VI-3.4.3.2 An Example

Consider I/O resources as a example (but note that the discussion is applicable to any resource managed by the scheduler).

I/O resources are divided into different classes and each class is assigned a priority; for example, terminals might have priority *10*, disks priority *9*, and communication lines priority *8*. (To keep process priorities less than or equal to *10*, all resources have priorities less than or equal to *10*).

A process begins executing at its base priority (say, *5*) and stays there until it blocks on an I/O resource (say, disks). While blocked, its priority is raised to the disk's priority (*9*). After the I/O, its priority is decreased linearly (by the same amount at each time slice) until it returns to its base priority (*5*).

As the process alternates between CPU usage and I/O requests, its priority fluctuates between its base priority and the priority of the I/O resources it requests (these may be different resources with different priorities). The process terminates at some priority level between its base priority and the priority of the I/O resource it last requested.

The presumption behind raising a process's priority to the resource's priority is that if the process issues one request for the resource, it is likely to issue another soon. The overall effect of the model is to favor I/O-bound jobs and penalize CPU-bound jobs, thus maximizing the use of system resources.

VI-3.5 Memory Scheduling

This section discusses memory scheduling.

Before a process can compete for CPU time, some of its instructions and data must be present in physical memory. (Invoking a job causes a series of faults that bring the program object, domain object, and other objects into primary memory; see Chapter VI-1). Thus, physical memory is as important a resource as the CPU, and memory scheduling is an important part of the scheduler.

The major goal of memory scheduling is to implement the *working set* model of memory management. The working set of a job is dynamically defined as the set of primary memory pages referenced by the job in the last time quantum, *T*, measuring backwards from a given time *t*. These are the pages which the job used most recently; identifying them and keeping them in memory reduces page fault rates and contributes to system efficiency. (See any standard operating system text for more information about the working set model).

Memory scheduling uses the following model:

- The system maintains a pool of free pages of primary memory.
- As long as there are enough pages in the pool, all the jobs in the mix are allowed to remain there and new jobs are allowed to enter the mix.
- To guard against the depletion of the pool, the scheduler periodically examines memory usage by all the jobs in the mix and transfers back to the pool any pages that are not in the working set of some job. This is done by examining each job's *Storage Resource Object (SRO)*. The SRO references a list of the pages each job has in primary memory. Any page that has not been accessed or modified in the last time quantum, *T*, can be returned to the pool. This is known as *SRO page replacement.*
- When the number of free pages in the pool falls below a *low water mark*, the scheduler tries to get more free pages by triggering SRO page replacement more often. If that doesn't succeed, the scheduler then pulls jobs out of the mix and releases their pages. The pages are given to the pool, and the jobs are swapped out to secondary memory. The scheduler keeps a waiting queue of swapped-out jobs.
- In order to achieve fair treatment for all jobs, the scheduler periodically examines the waiting queue and puts the job at the head of the queue in the mix. This ensures that no job starves while waiting for memory. The *aging* parameters in a job's SSO (`initial_age` and *age_factor*) determine the job's position in the waiting queue.
- The scheduler also periodically triggers global SRO page replacement, which attempts to free pages from the normal global SRO (pages in the frozen global SRO are not replaced).

VI-3.6 I/O Scheduling

I/O scheduling is done implicitly through the mechanism of resource-driven priorities, as described above.

VI-3.7 Summary

- The scheduler is a collection of hardware and software entities whose purpose is to schedule the execution of jobs (and thus processes).

- The scheduler's general objective is efficient use of system resources.
- The scheduler's task is to perform CPU scheduling, memory scheduling, and I/O scheduling.
- The type of CPU scheduling a job receives is determined by the SSO associated with the job when it is invoked. The SSO determines the job's service class, priority, time slice, memory type, initial age, and age factor.
- The scheduling daemon puts real-time, time-critical, and interactive jobs into the mix immediately, but puts batch jobs in a waiting queue until system load allows them to be put in the mix. Once a process is in the mix, its scheduling depends on several factors.
- The scheduling discipline for real-time and time-critical jobs is based on a fixed priority, but the scheduling discipline for interactive and batch jobs is based on dynamic, resource-driven priorities.
- The major goal of memory scheduling is to implement the *working set* model of memory management.
- I/O scheduling is done implicitly through the mechanism of resource-driven priorities.

Part VII
Type Manager Services

This part of the *BiiN™/OS Guide* shows you how to build *type managers*, software modules that implement new object types and their attributes.

The chapters in this part are:

Understanding Objects
Explains objects and their characteristics.

Understanding Memory Management
Explains how the OS manages memory.

Building a Type Manager
Shows you how to design and implement a simple type manager.

Using Type Attributes
Shows you how to define and implement type-specific *attributes*, packages or data structures supported by multiple object types.

Managing Active Memory
Shows you how to control object allocation and deallocation, and control object reclamation via garbage collection.

Building Type Managers for Stored Objects
Shows you how to design and implement type managers for objects stored on disk.

Understanding System Configuration
Explains how a BiiN™ node is configured as a collection of type managers that have configuration requirements. Each such type manager implements the *configuration attribute.*

Type Manager Services contains the following services and packages:

TM object service:
```
Countable_Object_Mgt
Global_SRO_Defs
Lifetime_Control
PSM_Trusted_Attributes
SRO_Mgt
Unsafe_Object_Mgt
```

TM transaction service:
```
Local_Transaction_Defs
Local_Transaction_Mgt
TM_Transaction_Mgt
```

TM concurrent programming service:
```
Job_Resource_Reclamation
Port_Mgt
Typemgr_Support
Unsafe_Port_Mgt
Unsafe_Semaphore_Mgt
```

configuration service:
```
Configuration
```

custom naming service:

```
Customized_Name_Mgt
Link_Mgt
Standalone_Directory_Mgt
```

backup service:

`Backup_Support` ***not implemented in this release***
`Trusted_Log_Mgt` ***not implemented in this release***

UNDERSTANDING OBJECTS 1

Contents

This chapter explains concepts related to objects and access descriptors. You can find most of this information elsewhere in the BiiN™ document set, but you would have to look in many different places. This chapter is the place where all pieces are brought together, so that you can understand the building blocks of the BiiN™ architecture.

The BiiN™ system has an object-oriented architecture; objects are the building blocks of the system. This is not the first system based on object-oriented programming. The difference between the BiiN™ system and other systems is the rigor with which object-orientation is implemented.

VII-1.1 Why Use Objects?

Objects are used in the BiiN™ system for the following reasons:

- Data abstraction
- Memory protection
- Secure and dynamic memory management
- Support for complex and extensible applications
- Uniform storage model for permanent and volatile memory
- Distributed storage model.

Each point above will be briefly explained in the following sections.

VII-1.1.1 Data Abstraction

In most cases your program will not be concerned with the inner workings of objects. An object appears like a black box to the programmer. The box has ''jacks'' and ''buttons''. As you press certain buttons the box takes things from the input jacks and sends something to its output jacks. Or the box performs some other operation. The two important points in the analogy are:

- The box's buttons do certain things and those things only.
- How the box performs its operations or how it looks on the inside is unknown. (See Figure VII-1-1)

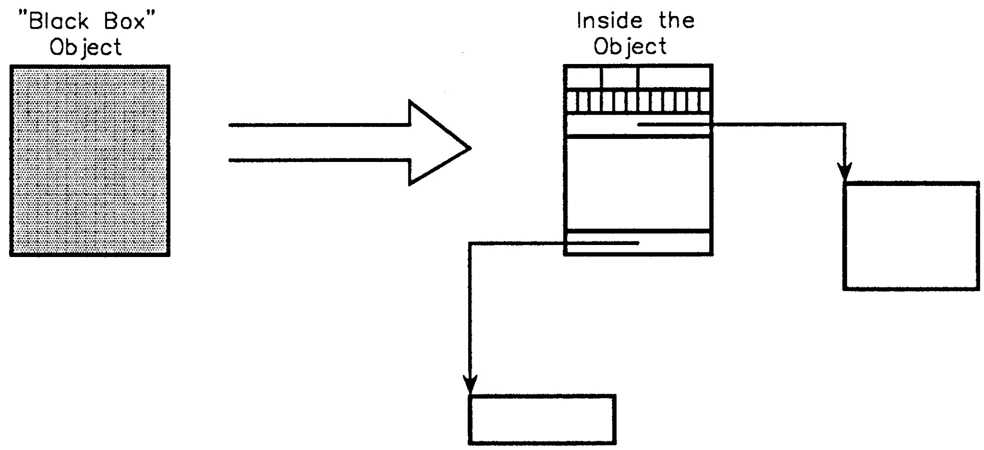

Figure VII-1-1. An Object as a Black Box

Objects present a well defined outside view. That means that their functionality is defined ''on their front panel''. How the object works is hidden from view. Data abstraction of this type has two advantages:

- A programmer can use an object without having to know what goes on inside just as you may use a television set without having studied the intricacies of electromagnetism.
- The inside of an object may be altered without affecting programs that depend on the outside of the object.

You can compare objects to Ada packages. The outside view of an object corresponds to the specification of the package. The representation of the object corresponds to the body of the package.

VII-1.1.2 Memory Protection

Objects are the unit of protection in a BiiN™ system. The memory of a BiiN™ system should not be viewed as an array of bytes but as a network of objects. The way the objects are connected can change at any time as the system runs. Each connection consists of a pointer called the *object index* and a list of access rights. These connections are called *access descriptors* (AD). The both provide and limit access. Connections can be made based on a strict ''need to know'' basis. Connections can only be made (ADs created) by the BiiN™ Operating System. The BiiN™ Operating System uses special hardware instructions to manipulate ADs. Every access to memory involves checking

- that an AD presented is a valid AD,
- that an AD has proper access rights,
- that the reference falls entirely within the referenced object.

While objects are protected by ADs, ADs are protected by the hardware. Special instructions are required to create and copy ADs. Nobody, not even the operating system, can circumvent this protection mechanism.

VII-1.1.3 Secure and Dynamic Memory Management

Objects are dynamic. They can be of any size from zero to four Giga bytes. They can be dynamically created, resized, and destroyed. Unneeded objects are automatically removed. For example you can create an object, change its size as many times as you want over the lifetime of the object and then simply abandon it. The operating system will pick up after you. Long running or very large programs can also explicitly control garbage collection. This relieves the operating system considerably.

VII-1.1.4 Support for Complex and Extensible Applications

Complex programs can never be entirely free of bugs. In a complex system a constant concern is that one program module not corrupt another. This problem is particularly hard to handle in conventional architectures: The instructions or data that have been corrupted may not even be related to the corrupting module.

This is a particularly acute problem when you want to extend important, sensitive, and complex applications, or maybe the OS itself. The traditional solution to the problem is to adopt a two-view scheme. In a two-view scheme the address space is divided into two levels, one level reserved for the operating system, and one level for the user. The interaction between the two levels is severely limited. The two-view scheme restricts functionality.

If address space is shared between user and operating system one risks major breakdowns of the combined system.

In the object-oriented architecture of a BiiN™ system addressing errors are confined to their origin: A wrong address will also always be an invalid address. This is done with a multiple-view scheme. Every application program, every system routine, in fact, ever job runs in its own protected address space. All jobs execute at the same level. The important ingredient in the multiple-view scheme is an efficient call/return mechanism that allows communication between protected address spaces.

For example, extensions to the OS run at the same level as the OS and are therefore able to use its full functionality. The same applies to applications. Any program can be easily extended without compromising reliability of the original program.

VII-1.1.5 Uniform Storage Model for Permanent and Volatile Memory

The BiiN™ system extends its model of protection and its object-oriented architecture to permanent storage. Objects in permanent memory (such as magnetic disks) are called *passive objects*. Objects in volatile memory are termed *active objects*. Permanent memory is termed *passive store*. There can be multiple active versions of an object but only one passive version at any time. In order to read the contents of an object or to write an object, the object has to be *activated* first. When a change to an object should become permanent, the object will be *passivated*. That means that either a new passive object will be created, or an existing passive version of the object will be updated. When multiple active versions of an object are present, the BiiN™ Operating System ensures that obsolete active versions cannot corrupt the passive object.

VII-1.1.6 Distributed Storage Model

Passive store is distributed -- spread over multiple BiiN™ nodes and transparently accessible from any node. One can view passive store as the glue that holds a distributed BiiN™ system together. Passive store is divided into volume sets. Passive objects are stored on volume sets. Along with each passive object, a *master AD* is stored on the same volume set. That passive AD contains a *unique identifier* (UID), unique for all times and on all BiiN™ nodes. Even if a disk is moved to another BiiN™ node or BiiN™ system, the passive objects stored on that disk will still be uniquely identified.

VII-1.2 How Objects Work

In the previous section you have learned what objects are, namely typed and protected memory segments. In this section you will learn how objects function in the BiiN™ architecture.

An object is characterized by a number of properties such as size, lifetime, type and a list of attributes. Objects can also be active or passive. In the following sections you will learn about these properties in more detail.

VII-1.2.1 Object Sizes

Objects can have sizes ranging from zero to four Giga bytes. Object sizes are rounded. (How object sizes are rounded is explained in chapter VII-5.) Objects can be created resized and destroyed at runtime (see Figure VII-1-2).

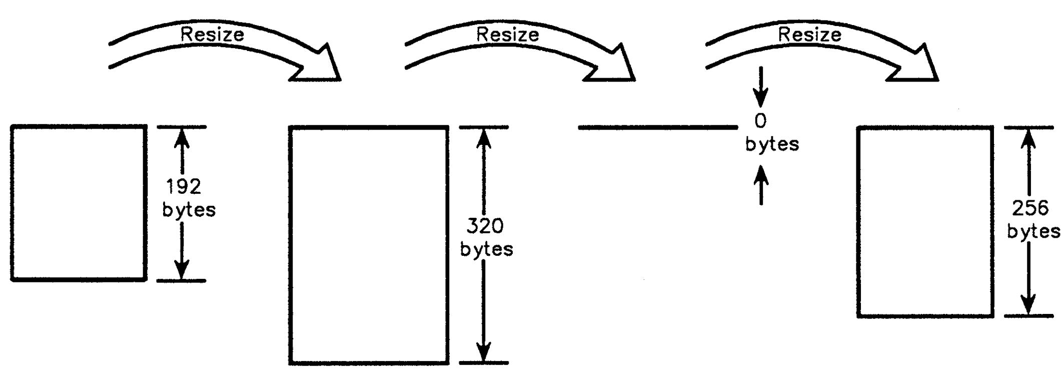

Figure VII-1-2. An Object Can be Resized

VII-1.2.2 Types

You probably know what typing is from programming languages such as Ada or Pascal. In one sense object types in a BiiN™ system are no different than data types in Ada. Since most of the BiiN™ Operating System is written in BiiN™ Ada, object types are implemented to a certain degree as Ada types. In another sense object types are very different from Ada types. Data in Ada is typed only at compile-time while objects are also typed at runtime. Whenever a software module attempts an operation on an object in a BiiN™ system, the OS first checks whether the operation is allowed for the object. While you can get around compile-time typing by using conversion functions or type overlays, there is no way to circumvent runtime typing

There are a number of predefined *system types* such as *disk, file, job* , or *program.* (For a complete list of system types refer to the Appendix of the *BiiN™/OS Reference Manual.*) On top, there is one peculiar type of objects called *generic* objects. Generic objects are untyped although, strictly speaking, they have a defined type, the so-called *generic* type.

You are not limited to the system types. Just as in Ada, you can define your own types and implement them on the system.

Object typing is complete and pervasive, more so than typing in programming languages. There are no backdoors that let you bypass the typing mechanisms.

VII-1.2.3 Object Protection

Typing protects an object from operations that are not defined for the object. There is another mechanism that protects the contents of the entire address space. This protection is provided by protected pointers called *access descriptors* (AD). As the name indicates, ADs provide access to objects. At the same time ADs limit access. Protection by ADs is complete. No object can be accessed without an AD. You can go so far as to identify an AD with the object.

Figure VII-1-3 illustrates the relationship between an object and an AD in a simplified way.

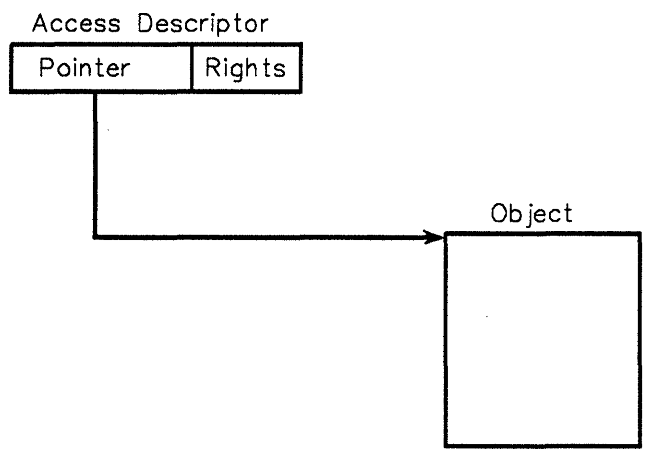

Figure VII-1-3. Object and Access Descriptor

VII-1.2.4 Attributes

While typing of objects serves two functions, namely protection and data abstraction, the same applies to attributes. Attributes are the means by which the prime capability of objects is realized; objects describe the operations that can be performed on them. An attribute is itself an object that acts as a label. The label typically describes an operation such as `Byte_Stream_AM.ops.Read`. All objects that allow `Byte_Stream_AM.ops.Read` carry a reference to this attribute. The mechanism works like this:

Objects have an attribute list that consists of *<attrib-ID,attrib-value>* pairs. The attribute-ID part references the attribute while the attribute value is typically an AD to a routine that implements the operation for the type.

All attributes contained in a particular object's attribute list apply to that object. In addition to these attributes an object inherits all attributes defined for its type. Those type-specific attributes are defined in the object's TDO.

For an example and an illustration of these dependencies see Figure VII-1-4.

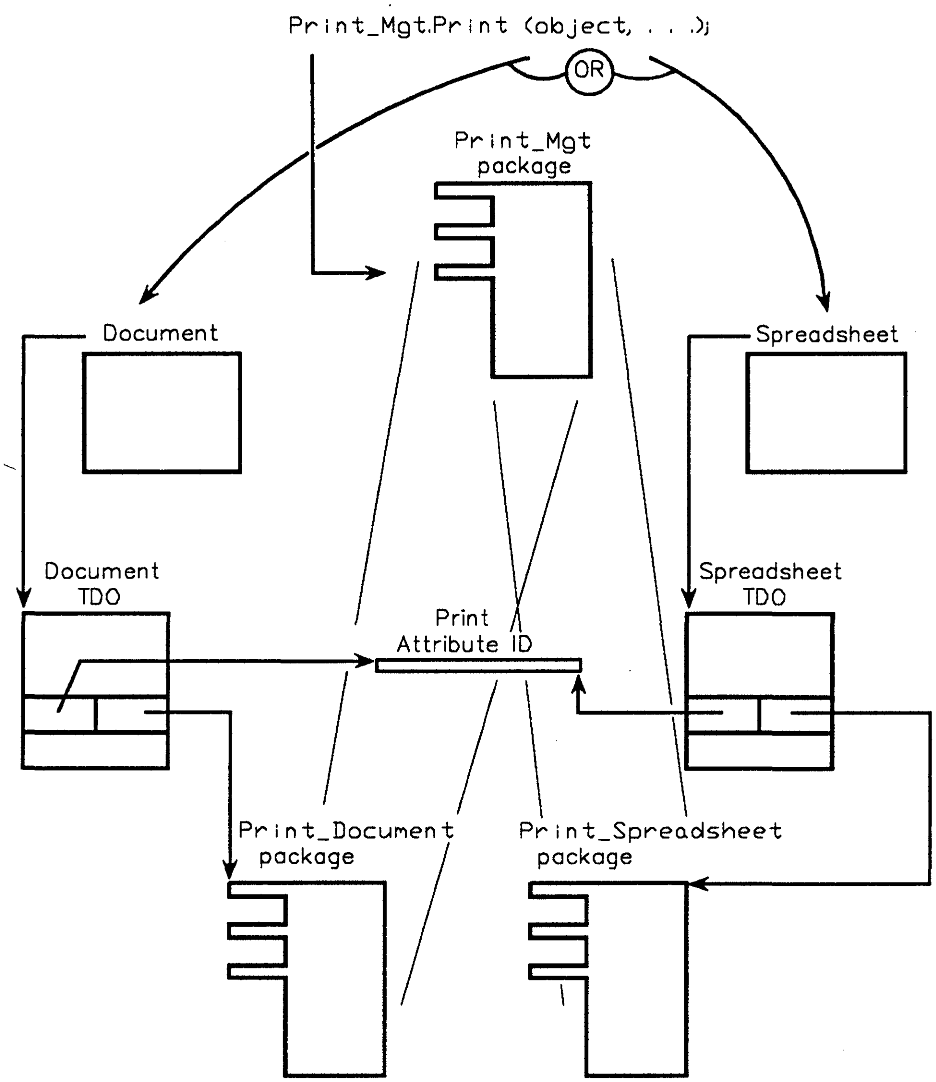

Figure VII-1-4. How Attributes Work

In Figure VII-1-4 there are two objects, a spreadsheet object and a document object. Both have inherited the attribute "`printable`" from their respective TDOs: The attribute lists of the two TDOs contain a reference to the same attribute "`printable`". The attribute values however are different: The `document TDO` has an AD to a package that implements printing of documents (named `Print_Document`) while the `spreadsheet TDO` has an AD to a package that is capable of printing spreadsheets (named `Print_Spreadsheet`).

Before concluding this section on attributes we shall briefly touch upon the general protocol of how attributes are implemented in a BiiN™ system.

Generally an implementor will establish a 1:1 correspondence between Ada attribute packages and attributes. There will be one attribute package for each attribute. The attribute package only contains subprograms and no other declarations. However, an attribute package can be nested inside another package that provides data declarations and subprograms common to all types. An attribute package must also have the Ada `package_type` pragma. This marks the package as an attribute package and binds it to the attribute ID, which is identified by its pathname. The body of an attribute package is empty.

As the next step, the implementor of an attribute will define various *instances* of the attribute package. These instances are the type- or object-specific implementations of the attribute package. In Figure VII-1-4 `Print_Spreadsheet` and `Print_Document` are such instances of one attribute package `Print`.

Instances have their own package specifications which all match the specification of the attribute package. The instances are bound to the attribute package by the `package_value` Ada pragma. Every instance has its own specific body and runs in its own domain. Instances cannot be merged into one domain with other packages.

VII-1.2.5 The Inside View of an Object

After having learned about the characteristics of an object, we proceed to explore how these concepts are implemented in the memory of a BiiN™ system. Figure VII-1-5 illustrates the inside view of an object. We have already learned about objects and ADs. Here we see that there are some more details to the picture:

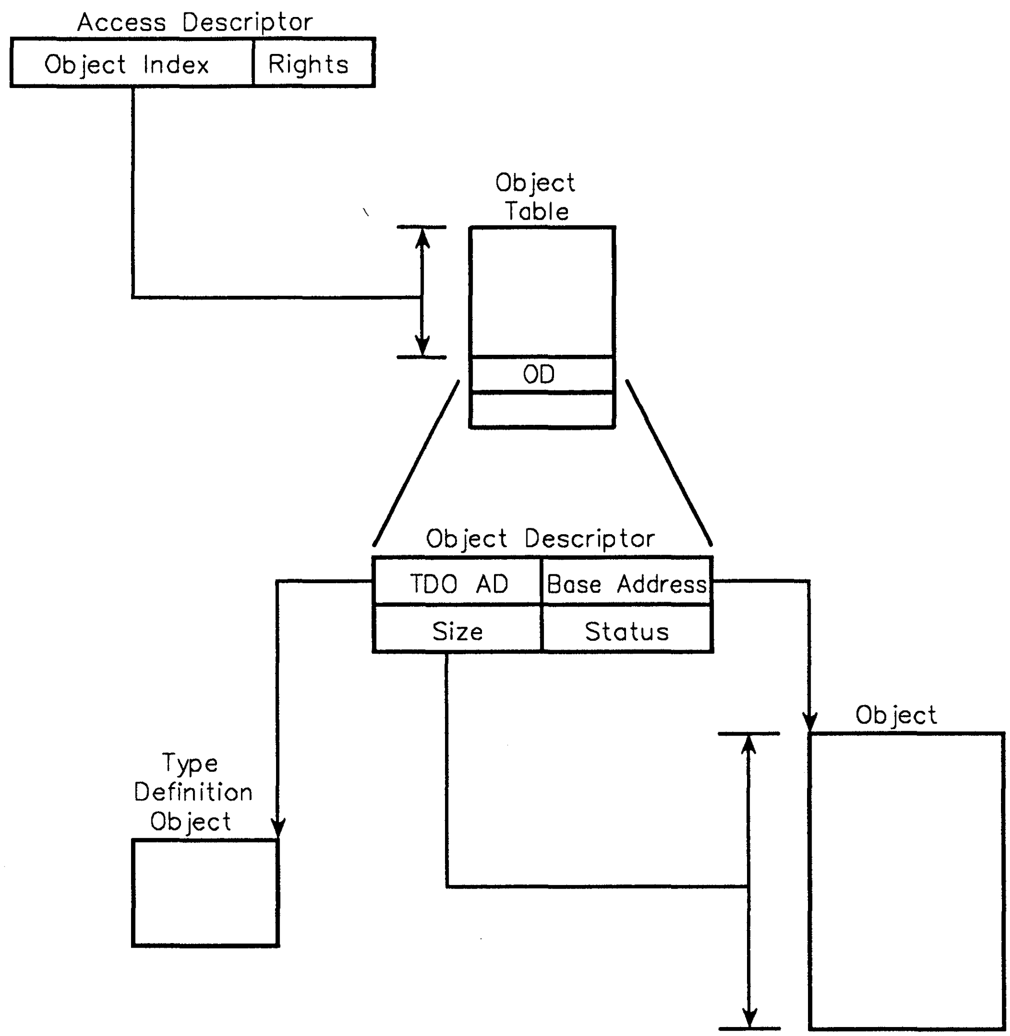

Figure VII-1-5. Objects Are Typed and Protected

An object consists of two parts, the object descriptor (OD) and the object's representation. When we talk of the size of an object, we refer to the size of its representation. The representation holds the contents of the object. The object descriptor on the other hand holds important information about the object, such as the physical address of its representation and its size. As Figure VII-1-5 indicates, an AD to an object points to the object descriptor not the object's representation. All object descriptors on one BiiN™ node are held in a one place, the *object table*. An object's representation may be moved around in memory by the BiiN™ Operating System but the object descriptor always stays in the same place.

The object's type is defined in the object descriptor by an AD stored there that points to a *type definition object* (TDO). There is one TDO for each distinct type. That means that two objects have the same type if their object descriptors reference the same TDO.

This model of objects with its two parts, object descriptor and object representation allows for a peculiar object, an object of length zero. Such an object has no representation and therefore really has zero length. This means that all information that pertains to the object is contained in the object descriptor. Objects of length zero are very useful as unforgeable identifiers. They can be compared to license plate numbers. The significance of a license plate number is not the information contained in it but the fact that it is different from all other license plate numbers.

VII-1.3 Address Space Protection

As software grows more and more complex, bugs become impossible to eradicate. No software engineer, nor any company can guarantee that their software products will not fail under any circumstances. Such software failures can have disastrous results as processors pervade our daily lives. It has therefore become imperative that failures be detected at their origins and that their influence be confined.

The most dangerous types of errors are addressing mistakes. By making such a mistake, a routine can corrupt data or programs anywhere in a computer's memory. Such a mistake may go unnoticed for a while until the corrupted data or programs are used. When the fault is finally discovered, it is almost impossible to locate its origins and prevent it from happening again.

Address space protection should not be monolithic as different programs require different levels of protection. A well tested routine running as a separate process would only suffer in performance if it had to drag along the same protection mechanisms that are needed for a recently implemented extension to the operating system.

The BiiN™ architecture provides a flexible and efficient protection scheme that addresses this problem. The unit of protection in a BiiN™ system is the object. An object is protected on three levels. (For an illustration, see Figure VII-1-6.)

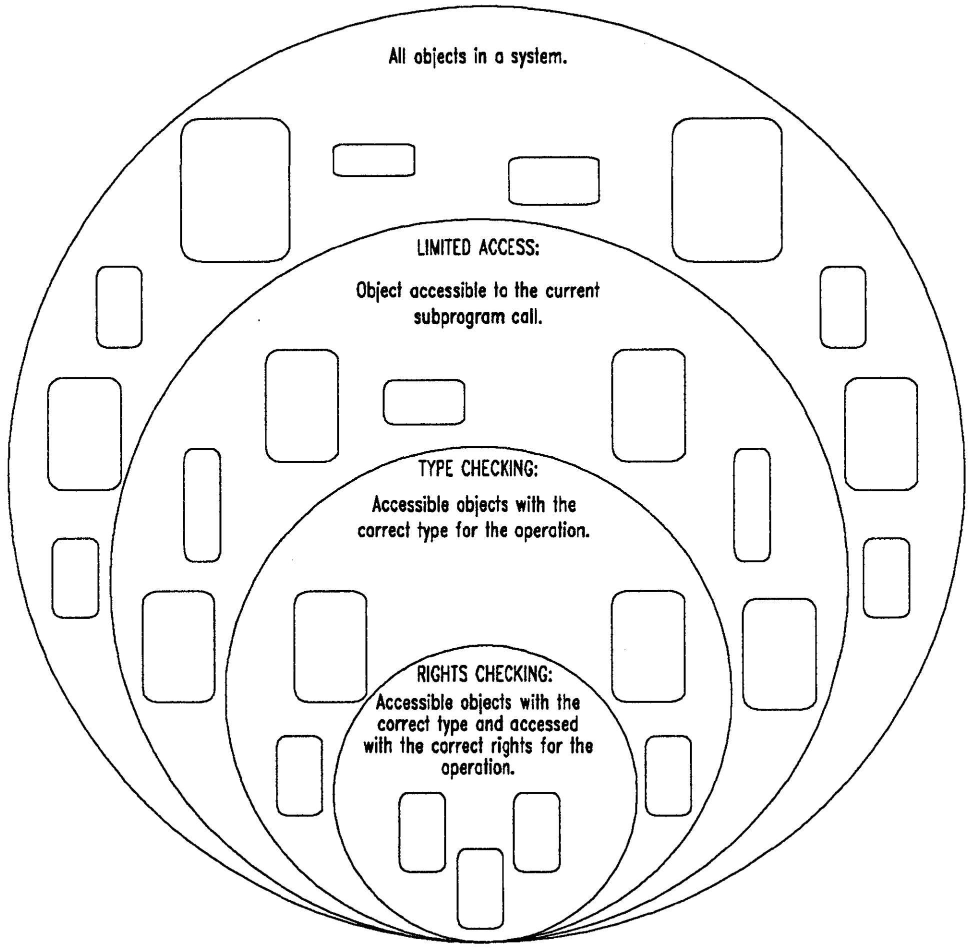

Figure VII-1-6. Threefold object protection

The entire memory of a BiiN™ system is organized in terms of objects. Objects can only be accessed by protected pointers, the access descriptors. An AD contains the information where the object it references is stored. But the AD limits the access to the object by way of *access rights* that are stored in it. Access descriptors are manipulated in controlled ways by the hardware. If a routine attempts to manipulate an AD, such as changing the address or tampering with the rights, the AD will automatically be invalidated. This is the basic protection that applies to all objects in a system.

ADs are given out on a strict "need to know" basis. Any subroutine therefore has access only to the objects that it needs to reference. Thus the set of objects accessible to any one call is strictly controlled. In Figure VII-1-6, this set is represented by the second outermost circle.

Objects are further protected by *typing*. Operations are tied to object types; an implementor defines what operations are permissible. This level of protection is represented in Figure VII-1-6 by the third outermost circle.

Finally the strictest protection is provided by the *type manager model*. A *type manager* is a routine that implements all operations on a certain type. Any routine that wants to perform an operation on the object protected by a type manager has to do so using a call to the object's type manager. This mechanism strongly confines any error that may occur in an operation on an object: Only the type manager can physically get to its objects. And only it is responsible for the objects' integrity. This level of protection is represented in Figure VII-1-6 by the inner-most circle.

In a BiiN™ system not all levels of protection have to be used at all times. Trusted routines can trade in protection for performance.

VII-1.3.1 Access Descriptors

Previously, we have characterized the memory of a BiiN™ system as a network of objects and access desscriptors as connections in the network. Access descriptor are protected pointers; pointers, because they contain a physical address; protected, because only the BiiN™ Operating System can create ADs. You may even identify an AD with the object because there is no way to get to the object except by AD.

Words on a BiiN™ system are 33 bits long. The 33rd bit of every word is a tag bit. If the tag bit is set, the hardware recognizes the word to be an AD. The information in an AD, address and rights together is 32 bits long. Figure VII-1-7 shows an AD.

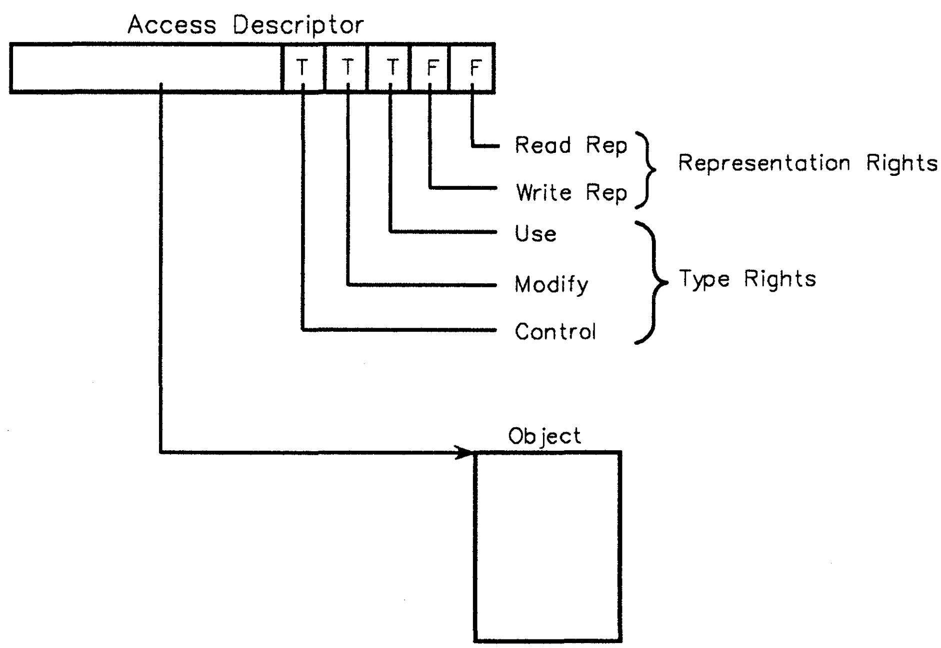

Figure VII-1-7. An Access Descriptor

The first 26 bits contain the object index, then a *local bit* follows, and the next 5 bits are the rights. (There can be 2^{26} different objects on one BiiN™ node at any time.)

There are five rights, three type rights and two representation rights. Type rights, as their name indicates are specific to object types. Their names may vary with the types they apply to. However, there is a naming convention for those three rights: They are called *use*, *modify* and *control*. In the case of a device, they may be renamed to *read*, *write* and *control* and in the case of a directory to *List*, and *Store*. There are no control rights in the case of directories.

Type rights give access to an object's logical structure. For example, if you have modify rights to a file you may write to this file record by record. Representation rights are different. There are *read* and *write* representation rights. They give access to an object's physical layout in memory. In the type manager model no routines are granted representation rights except the type manager. (See Figure VII-1-8)

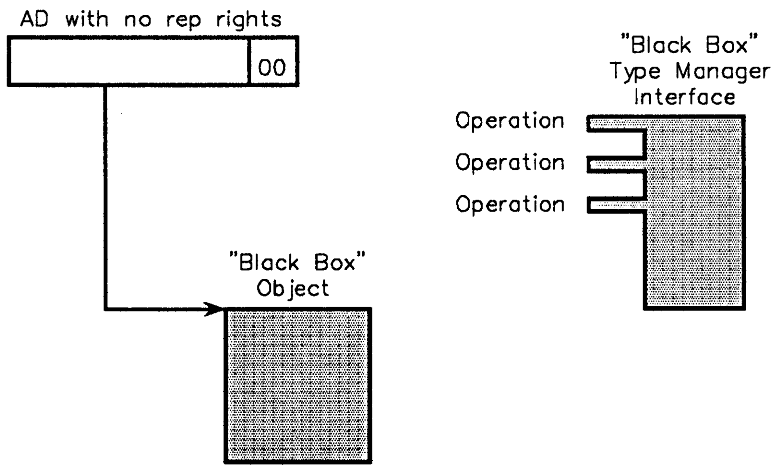

Figure VII-1-8. A Type Manager Makes the Object Appear as a Black Box

It is important to understand the difference between type rights and representation rights. For example, take read rights and read representation rights for a file. A file may have a very complicated layout in memory. It may sometimes be moved around by the operating system and it does not even have to be stored in a contiguous way. Having read rights you would never be aware of the way the file exists in memory. You could read the logical content of the file, however, and you could copy it. Having read representation rights to the file, on the other hand, you could read it bit by bit and find out precisely how it is stored in memory. Here we can go back to our black box analogy; type rights give you access to a black box's front panel. Representation rights are like a mechanic's license. They allow you to take a screwdriver, open up the box, and dig around inside.

VII-1.3.2 Type Managers

Type Managers provide the strongest protection in a BiiN™ system.

That protection is provided by the following mechanism: Any operation on an object protected by a type manager is a call to the object's type manager. The type manager is the only routine that operates directly on objects of its type: Only the type manager can create new instances of its type and only the type manager can remove those instances.

To use an analogy: In rare book libraries, users are not allowed into the stacks. Type managers act like librarians in such a library. Users of the library fill out request cards, and the librarians bring the books out of the stacks.

Type managers implement two paradigms of the BiiN™ architecture:

- Error confinement

- Independence of implementation details.

A well defined functionality is associated with objects of a given type. This functionality is provided by one module, the type manager. The type manager concept hides implementation details in the the type manager module and confines all errors to that same module.

As a new type is created, the system returns an AD for the type's TDO. That AD has *amplify* and *create* rights. It will be confined to the new type's type manager. A routine may now call the type manager and pass an AD with certain type rights to it. The type manager will use its AD to the TDO as a key and add representation rights to the passed AD. After performing the requested operation, the type manager strips off the representation rights and returns the AD to the calling program. By definition any routine that holds an AD with Create and Amplify rights to a TDO is a type manager for that type. ADs with representation rights should never be passed outside a type manager. There is is one exception to this rule; the rule does not apply to *generic objects.*

Generic objects are untyped in the sense that there is no type manager for generic objects. The operating system functions as the type manager for generic objects and gives out ADs *with* representation rights. Generic objects, however, are the only objects for which there are ADs with representation rights outside a type manager.

Generic objects are used whenever an untyped memory segment is needed. Representation rights are needed to write an untyped memory segment.

VII-1.3.3 Domains

Domains provide protected address space for program execution. A domain is represented by an object of type domain. How a program is split up over different domains is specified at link-time. The modules that make up a program may be linked into separate domains or some or all may be merged into one single domain. When calling a routine in a different domain address space is switched to the called routine's domain. Upon return, address space is switched back to the calling domain. The inter-domain calling mechnanism mutually protects caller and callee.

A separate stack may be associated with any set of domains. A set of domains that share one stack is called a subsystem. Subsystems are completely isolated from one another. The address space of a subsystem looks very much like an independent computer all by itself.

Figure VII-1-9 illustrates the details of a domain object.

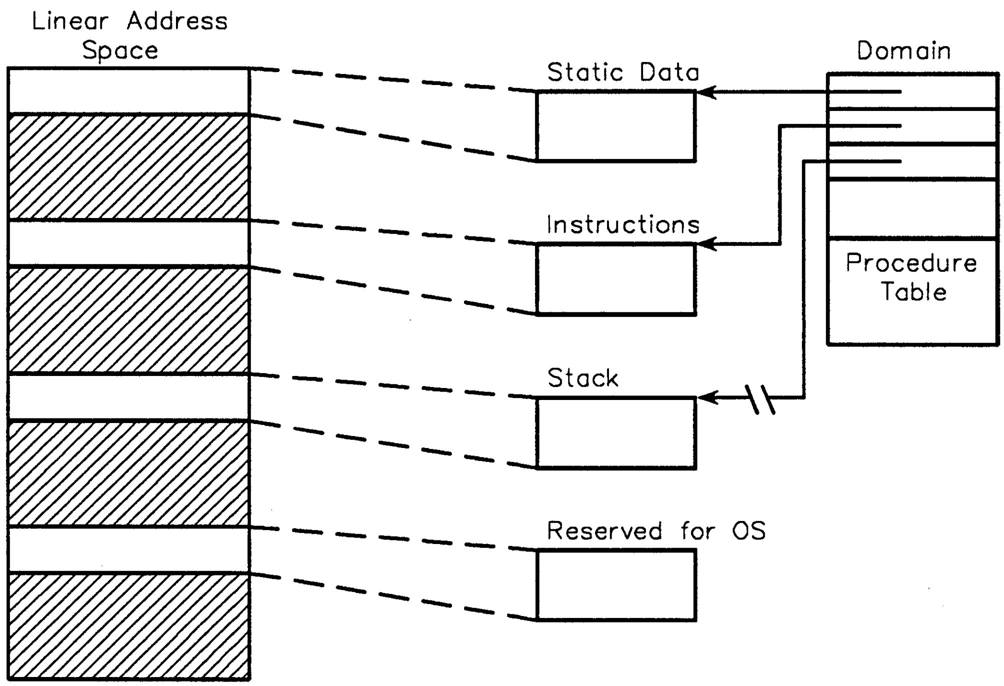

Figure VII-1-9. Linear Address Space and Domain

A domain holds ADs to the static data object, the instruction object, a subsystem ID, and an object reserved for use by the BiiN™ Operating System.

The static data object contains data that cannot be referenced outside the current domain. If a program has only one domain, the static data object contains all variables with global lifetime. The static data object also contains ADs to other domains whose external procedures can be called from this domain.

The instruction object contains the code for all subprograms defined in this domain.

The subsystem ID references a local stack object that contains parameters, local variables and housekeeping information used in subprogram calls. All domains in one *subsystem* and one job share a stack object. If you want to have a process executing with its own stack you have to put the process in its own subsystem.

There is a performance penalty attached to inter-domain calls. Only those modules that need the added protection should therefore be linked into separate domains.

VII-1.4 Passive Objects

We have mentioned before that there can be active and passive versions of an object. Most of our previous discussion applied to active objects. Although passive objects are very similar to active objects, there are a number of differences that you will need to understand. This section explains how objects act as the building blocks of *passive store*, a BiiN™ system's permanent memory.

VII-1.4.1 Active Memory

Active memory is the collection of objects in virtual memory on a particular BiiN™ node. An object can have versions in both active memory and passive store (Figure VII-1-10).

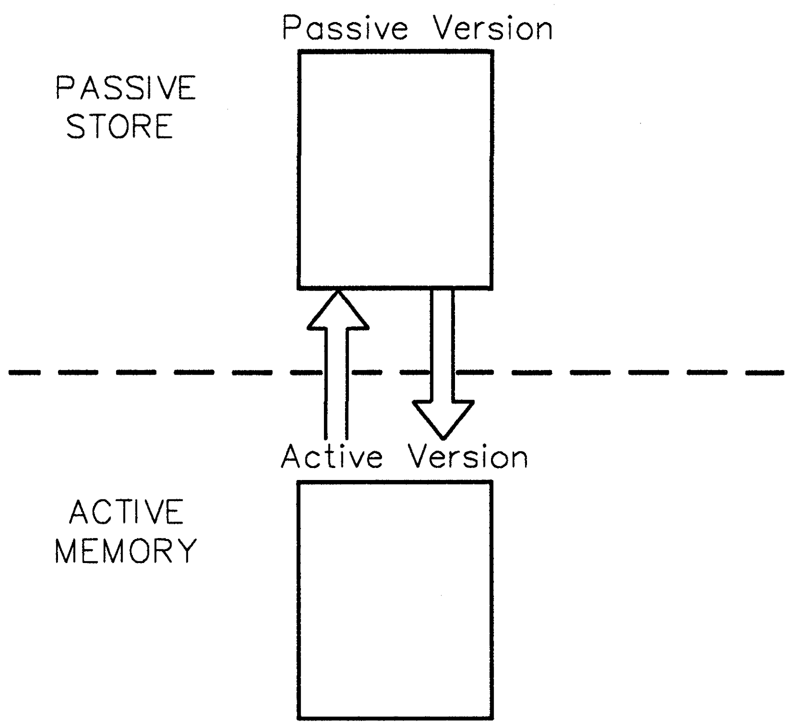

Figure VII-1-10. An Object's Active and Passive Version

Only active versions can be directly read or written. Reading or writing an object with no active version causes the object to be *activated.* Objects are activated on demand, transparently, just as pages of virtual memory are swapped in when needed. Both operations are invisible to your application. Changing an object's active version does not change the object's passive version. An explicit *update* call is needed to copy an object's active version to its passive version.

VII-1.4.2 Passive Store

While active memory is entirely part of one BiiN™ node, passive store is completely distributed in a BiiN™ system. Passive store is the glue that holds a distributed BiiN™ system together. (See Figure VII-1-11)

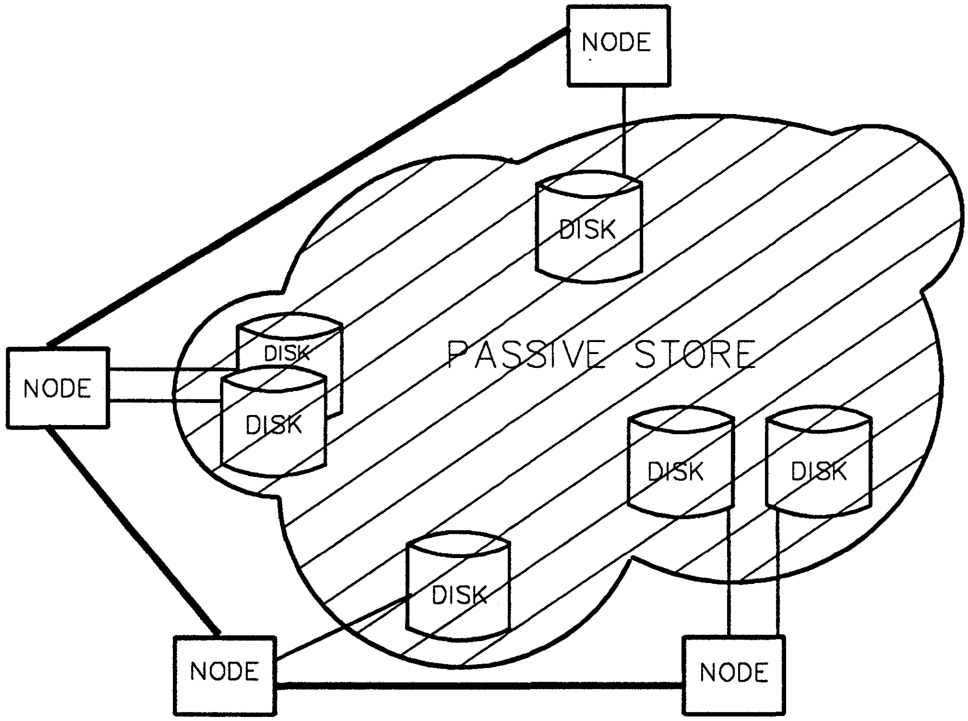

Figure VII-1-11. Passive Store Unifies All Nodes in a BiiN™ System.

Passive store wraps around an indefinite number of disks in a distributed BiiN™ system. Logically it is divided up into *volume sets*. Volume sets are associated with individual nodes. However, that association is transparent to the user.

VII-1.4.3 Passive ADs

When an object is first stored, passive store creates a passive AD for the object. A passive AD is a much bigger entity than an active AD. The reason is that a passive AD is a unique reference on an entire distributed system, while an active AD is valid only on a particular BiiN™ node.

Whenever an AD crosses the boundary between active and passive store or between different nodes of a distributed system, it has to be converted from its active to its passive form.

Just as there can be multiple active ADs to one object, there may be more than one passive AD to an object. (There may also be active ADs to passive objects.) One of the passive ADs is the *master* AD. All other passive ADs are called *alias* ADs. The master AD plays a crucial role. An object cannot be stored until a master AD exists. If there is no longer any master AD for an object that object will be removed. There are the following exceptions to that rule:

- If the master AD is stored in a directory and other directory entries on the same volume set reference the object. One of these alias ADs then becomes the new master AD.
- If the master AD is stored in another object and other ADs in that object reference the object. One of those alias ADs then becomes the new master AD.

VII-1.4.4 Passive Store Protection -- Authority Lists

Naming of and references to passive objects are slightly different than for active objects. The reason for this is simple: An AD once given out is irrevocable. That means that rights once granted by giving out an appropriate AD cannot be taken back. Generally this poses no problem in active memory since usually active objects only exist for short time periods. Objects on disk, however, exist indefinitely.

The model for protecting objects in passive store is different from the address space protection provided by ADs in active memory. Protection requirements are different for passive objects than for active objects.

In active memory a program should execute as much as possible in a secluded cell. Thus the segment of memory that can be affected by an erring program is kept to a minimum size.

This protection philosophy is inadequate for passive store for two reasons.

- Passive store is distributed. The view that any one job has of passive store should as wide as possible without opening up protection holes.
- Objects in passive store exist indefinitely. Information of who may access an object stays with the object. This allows the owner of the object to alter access over the lifetime of the object. (The philosophy behind active memory protection is to attach the information of who may access an object not to the object but to the requesting job. In this model it is difficult to revoke access once it has been granted.)

The difference explained in the second point above can be likened to the difference between a key lock and a combination lock. A key will always open the key lock just as an AD will always grant access to its object. But a combination can be made invalid when the lock is reset.

The protection provided for stored objects is based on the concept of an authority list. An authority list consists of *<ID, Type Rights>* pairs. When an object is first stored, an authority list can be specified by the storing process. If no authority list is given, the object will receive the default authority list of the directory in which it is stored. If there is no default authority list for the directory, the object receives the storing process's default authority list defined in the process globals. A passive object may also have no authority list.

An authority list is a vehicle for granting access to different users, user groups and programs. The owner can grant or revoke access at any time by specifying a new authority list. (Figure VII-1-12 shows how authority lists fit into the organization of passive store.)

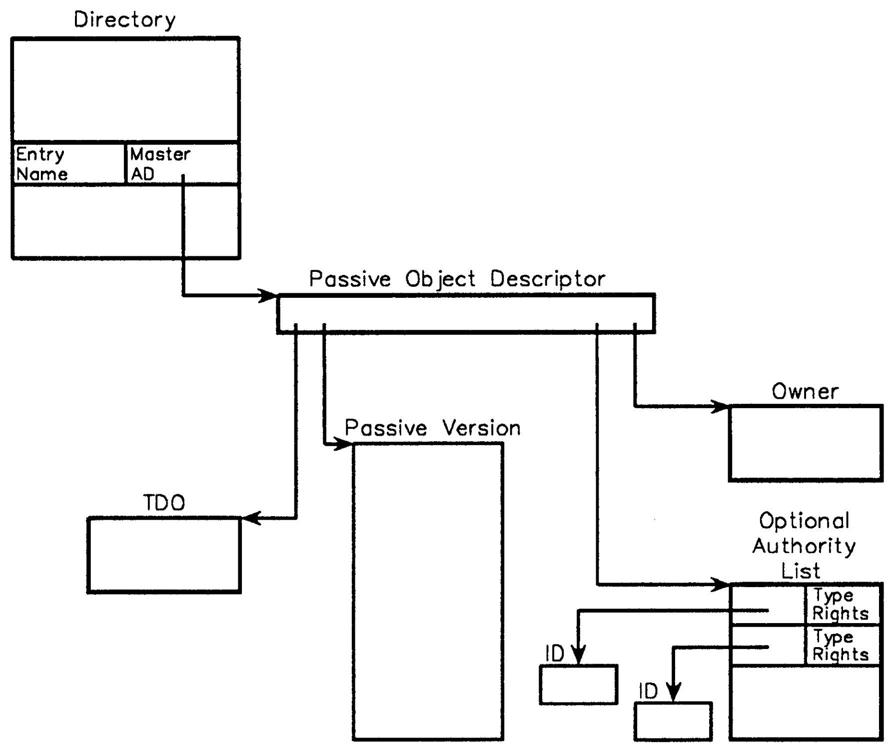

Figure VII-1-12. A Stored Object

Authority lists define access in two operations and for both in slightly different ways: Firstly, when a passive object is explicitly retrieved, the retrieving job's list of IDs is compared to the authority list and an AD is returned with the combined rights of all matching IDs. Secondly, when an AD is transparently activated, the activating process's ID list is checked against the authority list of the container and against the authority list associated with the AD proper. This ensures that stored ADs cannot be activated unless their rights are current. Should rights have been revoked since the AD was given out, the AD will loose those rights when it is activated. Note that an object's owner always has access to the object even if his ID does not appear in the authority list. For more details, see Chapter III-3.

VII-1.4.5 IDs

As you have seen in the previous section IDs are central to the protection concept used for passive store. It is therefore necessary to tell some more details about IDs.

IDs are maintained centrally in a a BiiN™ system, namely in the *Clearinghouse*. To get back to our previous example of the two different locks: Each ID is like the combination for a combination lock. (The analogy is a little bit weak at this point since combination locks usually only have one combination. Let's however disregard this for the moment and assume that there are combination locks that open by more than one combination.)

As IDs are the keys to stored objects, they in turn have to be protected. This is achieved by way of *protection sets* and passwords. Protection sets are similar to authority lists. They consist of *<ID, Rights>* pairs. The two rights defined for IDs are *portray* and *control.* The portray right grants the holder permission to add this ID to an *ID list.* Control rights allow the holder to alter the password on an ID. By specifying the proper password, one can obtain an AD to an ID with portray rights.

VII-1.4.6 Updating Stored Objects

Most calls to passive store are *transaction-oriented.* In particular, updates on stored objects can be included in a transaction. (A transaction ensures that all the operations included in it are executed as a unit: Either all the operations inside a transaction will be executed or none of them.) With the help of a transaction, you can prevent incomplete updates. Including calls to passive store in a transaction also prevents clashes between multiple jobs attempting an operation on the object. While the older of two transactions executes, it reserves the object. The younger transaction simply waits until the older one finishes.

Another problem arises when multiple active versions of an object exist. An obsolete active version could be used to update the passive version. Two situations can arise:

Multiple Activation Model:
: There are multiple active versions of a passive object. Passive store keeps track of all active versions and refuses updates from obsolete versions.

Single Activation Model:
: A *single activation object* is only activated in one *home job*. Other jobs that activate the object receive a token active version of the object called *homomorph*. Jobs that want to update the object have to communicate with the home job. For all operations on the object the job communicates with the home job of the object.

Both models are supported by the BiiN™ system. Depending on the needs of an application, the programmer can decide which one to use. In this context it is only important to note how updates are handled in these two models.

VII-1.5 Summary

After having read this chapter you should understand the following concepts:

- All information in a BiiN™ system is contained in objects.
- Objects are typed and protected memory segments.
- Objects are the unit of protection.
- Access descriptors are protected pointers. Objects can only be accessed with access descriptors.
- Objects can be dynamically allocated, resized, and destroyed.
- Objects may "know" what operations can be performed on them and how.
- Objects can have passive and active versions.
- Objects can be local to a job or global to a particular node.
- Passive objects are uniquely identified on all BiiN™ nodes and for all time.
- Access descriptors can pass freely between the nodes of a BiiN™ system.

If you understand all these concepts, you can go on to the next chapter which explains memory management.

UNDERSTANDING MEMORY MANAGEMENT 2

Contents

Objects are abstract constructs. Just as you cannot understand the concept of an automobile by studying metallurgy, you cannot understand objects by looking at their representation in memory. However, if you want to design a car, you will probably have to understand some metallurgy. Similarly, you will have to understand how memory is managed in a BiiN™ node if you are going to do some system programming, because objects are "made out of memory".

This chapter describes how a BiiN™ node manages its memory. It covers the underlying concepts of *virtual memory* and of the allocation and deallocation of objects. It discusses how objects are laid out in memory, when they can be moved around by the system and when not. And finally, it shows the forms of addresses in a a BiiN™ system and how they are resolved. This chapter does not give a detailed description of passive store. However, where passive store concepts are relevant to active memory management, they will be explained briefly. This chapter builds on the previous chapter (Chapter VII-1). You should either read that chapter or have a good understanding of objects and how they function in the BiiN™ architecture, before reading this chapter.

VII-2.1 Physical Memory Organization

Physical memory consists of a node's RAM and all disks that are mounted on the node. Physical memory is divided into *active memory* and *passive store.* Figure VII-2-1 shows how memory is organized in a BiiN™ system.

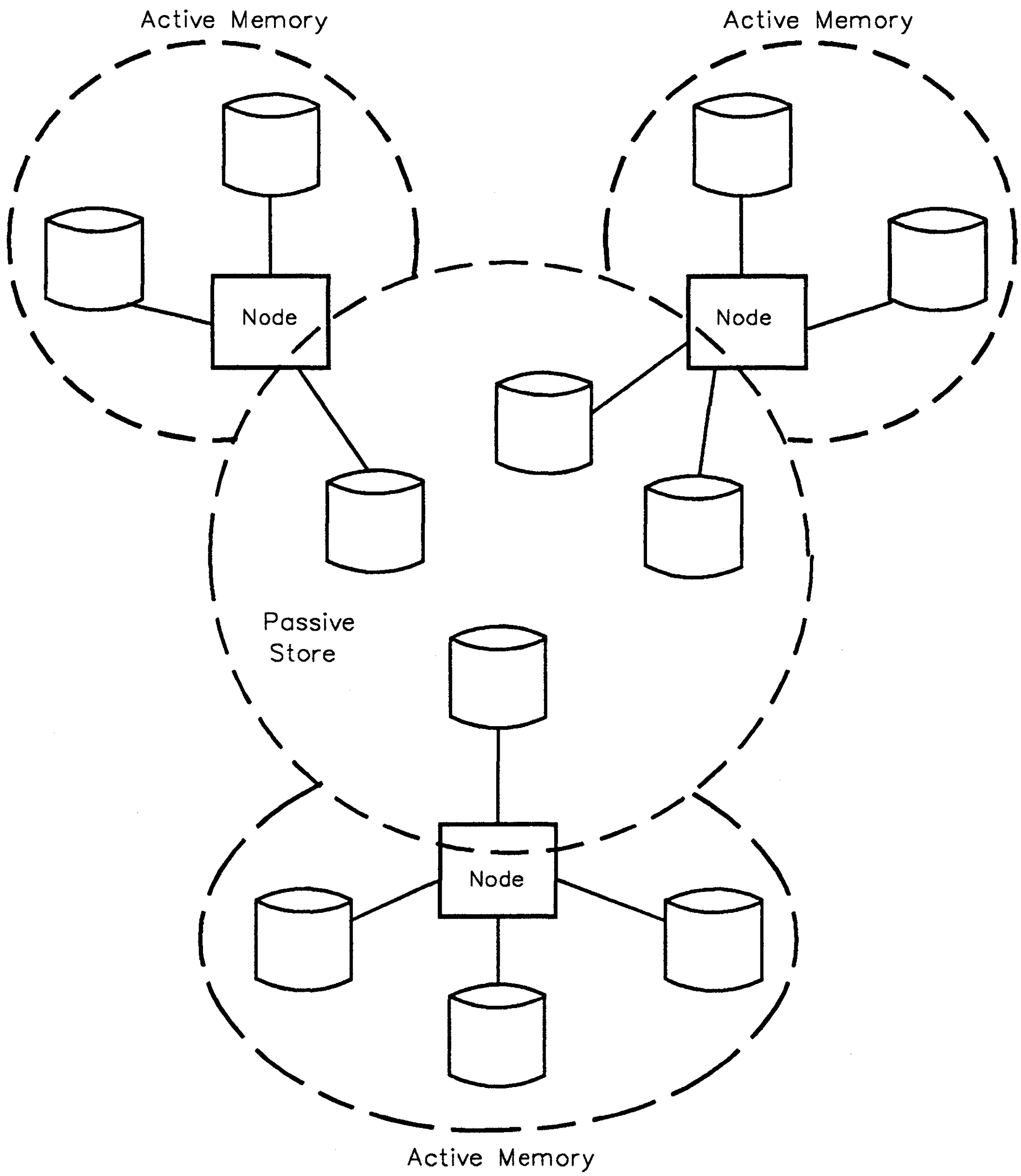

Figure VII-2-1. The Organization of Memory in a BiiN™ sytem

Active memory, as its name indicates, is the immediate ''working space'' of the processor. Active memory is also volatile. Its contents are lost whenever the system is turned off. Passive store on the other hand is permanent storage. Its contents cannot be lost unless a disk is damaged. (See Figure VII-2-2.)

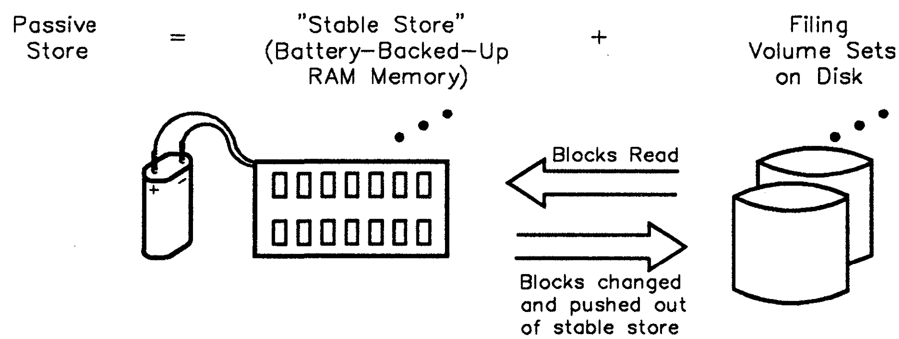

Figure VII-2-2. Passive Store

The memory pool on all disks of a node is partitioned into volume sets. Volume sets in turn consist of from 1 to 254 volumes. A volume set can span multiple disks. A single volume always resides on one particular disk. However, there can be more than one volume on a single disk. A volume set can be either a *swapping volume set* in which case it is part of the active memory, or a *filing volume set* and part of passive store. Swapping volume sets are invisible to the user. They appear as part of active memory, and from a user's point of view, the memory in a swapping volume set looks identical to the RAM.

The physical memory that underlies all other memory is partitioned into 4K byte page frames. Each page frame is uniquely identified by a page number. (See Figure VII-2-3.) A page frame is simply an empty page. A page is the unit of abstraction of memory management. The smallest unit that memory management recognizes is 64 bytes.

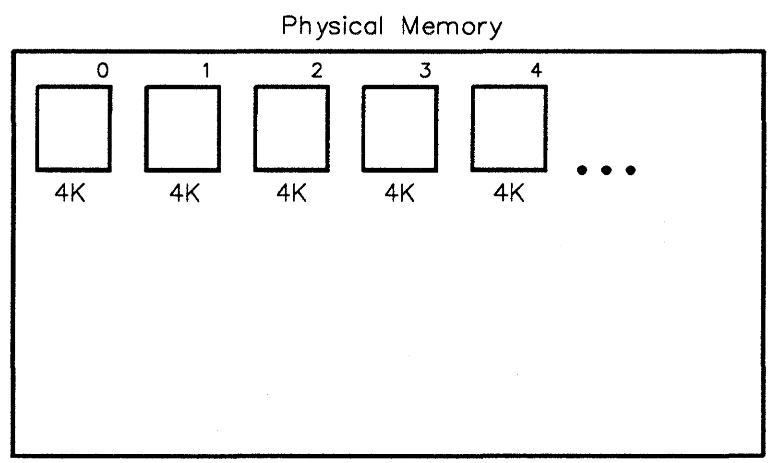

Figure VII-2-3. Physical Memory is Divided into Pages

Private to memory management is a central *page frame table* (PFT) where information about the contents of all page frames is stored. Since a single page frame may contain different information as time progresses, the contents of the *page frame table entry* will change as well. (There is a parallel here between physical and logical memory organization: Object table and page frame table and object descriptor (object table entry) and page frame table entry play similar roles. An important difference between the two is that the object table is recognized by the hardware, while the page frame table is purely a software concept.)

VII-2.2 Virtual Memory Organization

Active memory is organized according to the *virtual memory concept*. This means, the part of memory that is directly accessible to the node may span parts or all of the node's RAM and mass storage devices such as disk drives as well. The processor's total physical address space is 2^{32} bytes. (That is about 4G bytes.) (See Figure VII-2-4.) The total virtual address space permissible is 2^{58}bytes, consisting of 2^{26} objects and 2^{32}bytes per object. The virtual memory concept frees the system from the limitations imposed by relativley scarce primary memory.

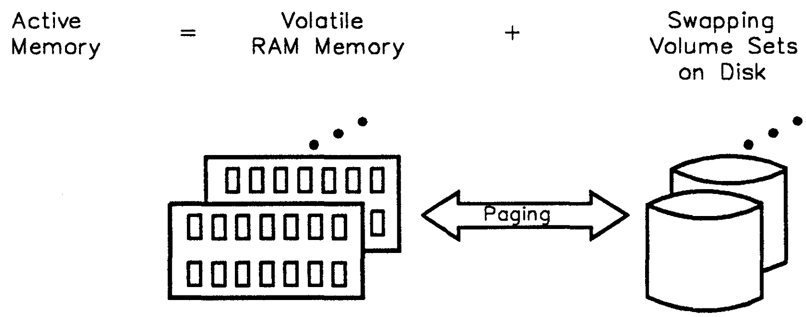

Figure VII-2-4. Active Memory Uses Both RAM and Disk.

Virtual memory management takes advantage of the fact that the entire address space of the node is not used simultaneously at all times. The processor can only directly address pages that are available in RAM. This part of memory is called *primary memory*. Memory management moves pages in and out of primary memory in such a way that the user has the illusion that all the information is contained in primary memory. Pages are swapped in as they are referenced and swapped out when they are no longer needed. A page is either *accessible* or not. If the page is accessible, it means, the page resides in primary memory and the process can get to it directly. If the page is not accessible, memory management retrieves it from its location in *secondary memory* (on disk, in the swapping volume set) and places it in primary memory.

There is a *common page pool* that is a list of free pages in primary memory. When a job requests space in RAM, pages from the common page pool are allocated to it. When a page that is not altered is returned to the common page pool, then, if a process references the page, it can be reclaimed from the pool, thereby avoiding a swap-in. In essence, the common page pool represents a cache of pages in the swapping volume set If a page is not available in the common page pool, it is swapped in from disk. That means, its contents is copied into a newly allocated page frame.

VII-2.2.1 The Object Table

Physical memory is organized in terms of pages. On the other hand logical organization of memory is in terms of objects. The page frame table (PFT) centralizes important information about pages. Analogous to the PFT in the organization of physical memory is the object table in the logical organization of memory. (The object table is a hardware defined and hardware recognized data structure, while the page frame table is a purely software defined data structure.) The PFT consists of page frame table entries, and the object table consists of object descriptors. (See Figure VII-2-6.)

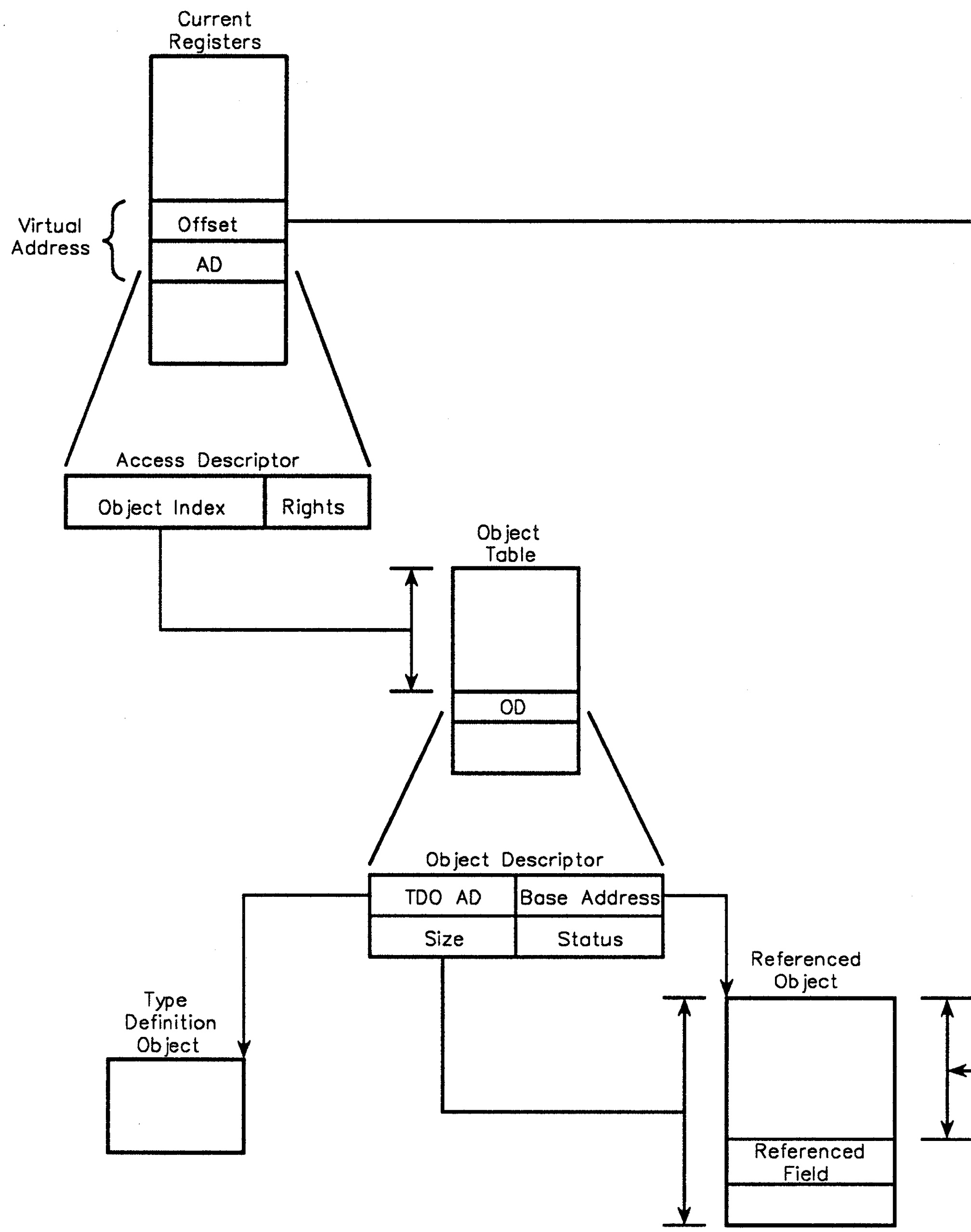

Figure VII-2-5. The Object Table and Object Based Adress Translation

Objects can only be referenced by access descriptors (ADs). There can be a multitude of ADs to any single object. It is necessary to have one single place where important information about the object is stored, such as its physical address. Otherwise all ADs to the object would have to be updated if some of the information changes. For this reason, there is exactly one object table per node.

VII-2.2.2 Object-Based Address Translation

Figure VII-2-5 also illustrates the addressing mechanism. The BiiN™ system recognizes two types of addresses, *linear* and *virtual* addresses. Linear addressing is faster than virtual addressing, but is restricted to a single domain. Linear addresses are used for programs that execute entirely inside a linear address space. This would typically be the case with FORTRAN and Pascal programs. In order to access arbitrary objects in the system you *have* to use virtual addresses. Figure VII-2-6 shows a valid virtual address.

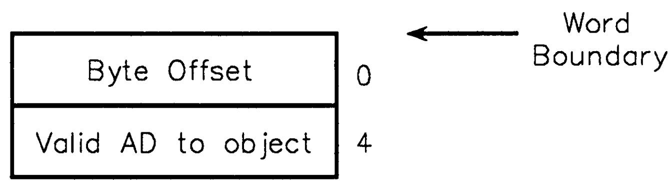

Figure VII-2-6. A Valid Virtual Address

Virtual addressing is an object-based addressing scheme. Figure VII-2-5 illustrates the virtual addressing scheme. A virtual address consists of two parts, an AD to the object that contains the field that you want to access, and an offset into the object that specifies where the field is located inside the object. A linear address is an offset by itself, witout an AD.

As mentioned previously, the AD does not reference the object directly but rather it refers to the object descriptor in the object table. The object descriptor holds the physical address of the object.

VII-2.2.3 Storage Resource Object

There is one *storage resource object* (SRO) associated with each job. It represents a pool of storage local to the job and all its processes. When an SRO is first created, a certain *storage claim* is assigned to it. As storage is allocated from the SRO the storage claim is debited, and if storage that had been allocated from the SRO is deallocated, the claim is credited with the proper amount. A job's local SRO is a global object which is removed once its controlling job terminates. In addition to local SROs there are two global SROs for each BiiN™ node, one controlling *normal memory* allocation and the other one controlling *frozen memory* allocation. Global SROs can only be referenced by administrative users and trusted type managers. Global SROs have unlimited storage claims. SROs are *active-only* objects: That means that SROs cannot be passivated. (For a discussion of normal and frozen memory, see section VII-2.2.5.) Figure VII-2-7 illustrates SROs in a node's virtual memory.

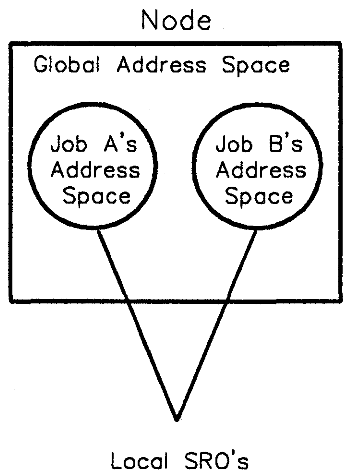

Figure VII-2-7. Active Virtual Memory, Jobs, Nodes and SROs

VII-2.2.4 Object Representations

An object's representation is an area in virtual memory that holds the contents of the object. An object's representation has a certain size that can range from 0 to 2^{32} bytes. However, object sizes are rounded depending on the size of the object:

1. If `size = 0` bytes, or if the object is a *semaphore*, then the object's representation is entirely contained within the object descriptor. These objects are called *embedded objects.*
2. If `0 < size <= 4K` bytes, then `size` is rounded up to the next multiple of 64 bytes. These objects are called *simple objects.*
3. If `4K < size <= 4M` bytes, then `size` is rounded up to the next multiple of `4K` bytes. These objects are called *paged objects.*
4. If `4M < size <= 4G` bytes, then `size` is rounded up to the next multiple of `4M` bytes. These objects are called *bipaged objects.*

The reason for the rounding outlined above stems from the paged structure of the underlying physical memory. The following paragraph outlines the mechanism. For more details refer to *BiiN™ Systems CPU Architecture Reference Manual.*

Simple objects can share a page frame with other simple objects. If an object's size is equal to `4K` bytes, it will occupy a page all by itself. In the case of a paged object the object descriptor references a *page table* (PT). A page table is simply a list of all pages that are part of the object's representation. The page table is located on a page frame itself, possibly together with other object's page tables. If a paged object's size is equal to `4M` bytes, the page table will occupy an entire page by itself. The object descriptor of a bipaged object references a *page table directory* (PTD). This is a list of page tables which in turn are lists of page frames. Instead of having one very long page table there are two levels of page tables (hence the name bipaged objects) -- many 4K page tables, and one level up, a table of those page tables. In the extreme case of an object occupying `4G` bytes, the page table directory itself occupies an entire page.

The object table is a paged or bipaged object. It is handed out in units of single pages which can contain up to 256 object descriptors. Whenever possible, the object table is kept down to a paged object to keep down address translation times. Only when necessary will the object table become bipaged.

VII-2.2.5 Frozen and Normal Memory Types

In certain cases, such as real-time or time-critical applications the virtual memory mechanism of swapping pages in and out of primary memory may cost too much time. Upon request, a job can run in *frozen memory*. The job's SRO will then allocate objects that will not be moved between primary and secondary memory but will reside entirely within primary memory. A local SRO that has a *frozen memory type* has an infinite storage claim. The designer of the application will have to take care that there is sufficient primary memory to run the program. Furthermore, in order for all pages to be allocated before the program runs, the user must have *allocate-on-creation* rights for the SRO.

Most other programs will run in *normal memory*. They have an SRO with a *normal memory type*. The SRO than has a given fixed storage claim.

VII-2.3 Different Allocation Policies

Two policies are used when paged objects are allocated in primary memory. The standard policy for SROs with a normal memory type is *allocate-on-reference*: First, only the page table directory is allocated for a bipaged object and the page table of a simply paged object. Second level page tables of bipaged objects and pages of paged objects are physically allocated in memory only when they are directly referenced.

The second policy, called *allocate-on-creation*, is reserved for SROs with frozen memory type. The SRO also needs to have *allocate-on-creation-rights*. Allocate-on-creation can be explicitly enabled and disabled for such an SRO. If an SRO with allocate-on-creation enabled allocates an object, the entire representation of the object will be allocated. This technique is useful for time-critical and real-time applications.

VII-2.4 Object Lifetimes

There are *local* and *global* objects in the BiiN™ system. Local objects are local to a particular job. That means that the active version of a local object is removed when the controlling job finishes.

A local object can however be passivated, and the passive version will survive when the controlling job finishes. When the passive version is again activated, its active version will again be a local object and will automatically disappear, once the job that activated the object finishes. A local object that has never been passivated will disappear completely once its controlling job finishes. Global objects exist outside any particular jobs. There are two types of global objects, *unbounded* global objects and *countable* global objects.

An unbounded global object's active version can exist indefinitely, or more precisely, until it is explicitly removed by global garbage collection. Global objects can also be passivated and thus survive system crashes and explicit garbage collection.

Countable global objects behave very much like unbounded global objects. However, unbounded global objects have one distinct disadvantage that countable global objects avoid: Unbounded global objects can only be removed by global garbage collection. Global garbage collection is a very expensive process because it may involve extensive disk traffic. It is desirable that it not be used too often. Countable global objects can be deallocated without global garbage collection. This is done with the following technique.

For countable global objects, there is a mechanism that keeps track of all references to a particular object. Whenever an AD is given out to a job for the first time, the reference count is incremented by one. Also, whenever a job terminates that held an AD to the countable global object, the reference count is decremented by one. If the reference count equals zero, object management is notified and then removes the object. Note that the reference count keeps track of how many jobs hold references to the object, not how many ADs have been given out. A job can also *logically delete* its AD to an object. The job then continues to run but forfeits its access to the particular object. This causes the count of logically deleted references to be incremented. When the count of logically deleted references is equal to the reference count, deletion of the object also results. The BiiN™ Operating System and the hardware work together to prevent lifetime violations.

ADs can also be local and global. On the simplest level, this means, ADs to a local object will always be local ADs. If this were not so, global ADs to a local object could outlive the object. For that same reason local ADs are confined to one job. Global objects can have local and global ADs. Countable global objects, however, have only local ADs. This ensures that all ADs that belong to one job will disappear once the job terminates.

VII-2.5 Object Deallocation Strategies

There are various ways of removing, or deallocating, objects that are no longer needed. This is an important task. Without it, memory would be exhausted in a very short time period. The way objects are deallocated depends on the object and on the needs of the job that uses them. In particular, there are these methods for deallocating objects:

- Explicit Deallocation
- Local Garbage Collection
- Global Garbage Collection
- Reference Counting
- Deallocating Passive Versions.
- Job Termination

Explicit deallocation (using `Object_Mgt.Deallocate`) is the simplest, most direct method to remove an object. It is used whenever a job "knows" that an object that it has created is no longer needed. Note, however, that such deallocation removes only the object's active representation. The object descriptor will still be there. If an AD is used to access an object whose representation has been deallocated and which has no passive version, the exception `System_Exceptions.object_has_no_representation` is raised. If there exists a passive version of the object, it is transparently activated. Note, however, that when you deallocate an object's representation, the object's passive version is not updated automatically. If you want to save any changes on the object, you have to specifically update the passive version.

There is an operation available to trusted routines called `Unsafe_Object_Mgt.Unsafe_deallocate`. This operation removes not only the object's representation but the object descriptor as well. This operation is unsafe because if there are any ADs to the object after the object has been completely removed from the system, a use of this AD will result in a dangling reference. A routine that uses `Unsafe_deallocate` has to ensure that there are no ADs left to the object outside the routine itself. Failure to do so can cause fatal system behavior.

Local objects for which there are no more ADs can be reclaimed by local garbage collection. The purpose of local garbage collection is to enable long-running jobs to periodically clean up their address spaces. Garbage collection can be started and then runs as a daemon. When run as a daemon it will wake up periodically whenever the storage claim of the job falls below a certain adjustable percentage. A minimum delay between runs of the garbage collector (GCOL) can also be specified. This is to prevent GCOL from running permanently when a job's storage claim becomes low.

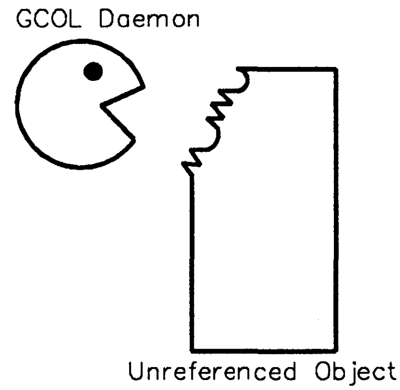

Figure VII-2-8. Garbage Collector

GCOL finds each object with no reference and labels it as garbage. It then starts to remove these objects. Differently from an explicit `Deallocate`, GCOL also removes an object's object descriptor. It can do so because it has previously made sure that no ADs to the object exist.

When a job finishes all objects local to the job are removed completely, representation, local ADs, and object descriptors.

Besides the local garbage collection, there is also a global garbage collection mechanism. Global garbage collection works for global objects the same way local garbage collection works for local objects. Global garbage collection is invoked periodically by the system and removes all unreferenced objects. Global garbage collection is an expensive process: It may involve a lot of disk traffic. Therefore, global garbage collection should run as infrequently as possible.

As mentioned previously, countable global objects can be removed without the overhead of garbage collection.

VII-2.6 Controlling and Accounting for Memory Resources

Jobs are dispatched to the processor by a scheduler. The scheduler recognizes four different classes of jobs: *batch, interactive, time-critical* and *real-time.* What class a particular job belongs to, depends on what SRO the user specifies when the job is started. (A user has to have the necessary rights to an SRO in order to run a job from it.) Depending on the type of the job, a storage claim of a certain size is defined in the job's SRO by the scheduler.

When an object is allocated from an SRO, the job's storage claim is charged. Accounting is done for the number of object descriptors allocated from the SRO and for the size of the representation of the object. If a local SRO gets to the bottom of its claim, local garbage collection is automatically invoked. In most cases this will result in enough memory space being reclaimed to be able to satisfy the job's allocation request. However, if the garbage collection cannot reclaim enough space to handle the job's allocation request, the job is terminated with a message that states that resources have been exhausted. Accounting is done on a per job and per node basis.

In addition, the class of a job has a more subtle influence on memory allocation than just setting upper limits on the allowed space. In particular, it specifies whether a job is subject to virtual memory paging or not. In the extreme case, a job can run in frozen memory. That means, all of its virtual memory is primary memory. Thus all the job's objects are immediately accessible without swapping pages. This increases performance considerably.

VII-2.7 User-Transparent Memory Management Functions

Most of the functions of memory management are executed transparently to the user. In particular this includes the following:

- Object Activation
- Virtual Memory Paging
- Global Garbage Collection
- Compaction
- Optimized Handling of Instruction Objects.

VII-2.7.1 Object Activation

This section describes the mechanism behind transparent object activation. Typically, an object's representation is deallocated and a process holds an AD to the object. When the process touches the object, the BiiN™ Operating System finds that the object has no representation. At that point it attempts to find the object in passive store. If it succeeds, the passive version is copied into active memory and becomes directly available to the requesting process. Otherwise, activation fails.

VII-2.7.2 Virtual Memory Paging

The virtual memory concept solves the problem that primary memory is scarce. A large part of virtual memory is secondary memory; that is disk. When a process touches a page that is presently held in secondary memory it will be swapped into primary memory. Secondary memory that is part of virtual memory is called *swapping memory*. Swapping memory is devided into volume sets, just as passive store. Swapping pages between swapping volume sets and primary memory is invisible to the requesting processes. Extensive page swapping, however, slows down program execution. For that reason real-time jobs have all their memory requirements satisfied in primary memory. (In this case the programmer has to make sure that there is enough primary memory available to satisfy the job's demands.)

VII-2.7.3 Global Garbage Collection

The system periodically invokes a global garbage collector daemon. The daemon is responsible for cleaning up a node's global memory. It removes all global objects for which no AD exists on that node. Garbage collection runs in the background and is invisible to the user. Global garbage collection involves a great amount of overhead. This is because the objects that garbage collection is looking for are unreferenced objects. Objects that have not been referenced in a while tend to move to secondary memory. Finding all those objects and removing them involves a lot of disk traffic. Remember also that garbage collection has to search all objects on a node for references.

VII-2.7.4 Compaction

The representation of a simple object usually takes up less than one page of of memory (4K bytes). When pages are swapped out, compaction is transparently invoked. Compaction takes simple objects and optimizes memory use by placing multiple simple objects on one memory page. Swapping always happens page by page. When a user requests a simple object that is presently on a swapping volume set and shares a page with other simple objects, the entire page that holds the object is swapped in.

VII-2.7.5 Optimized Handling of Instruction Objects

As their name indicates instruction objects hold processor instructions and constants necessary for program execution. Program execution is optimized in three ways:

- Pages of instruction objects are directly paged in from the file. You do not need to explicitly activate (or load) the instruction object.
- The representation of a (local multiple activation) instruction object is physically shared by all jobs using it whenever possible. This avoids having multiple identical copies in active memory.
- When a job terminates, pages of the instruction object may remain reclaimable for some time. That means, another job that runs later and uses the same instructions can reclaim those pages without having to copy them from disk.

VII-2.8 Summary

After having read this chapter you should now have a basic understanding of how active memory is managed in a BiiN™ node. In particular, you should have grasped the following concepts:

- Physical memory organization
- Virtual memory
- The object table
- Storage resource object

- Objects representation
- Granularity of object sizes
- Memory types
- Object allocation
- Object lifetimes
- Object deallocation
- Control of memory resources
- Transparent memory functions
- Addressing

BUILDING A TYPE MANAGER 3

Contents

A *type manager* is a program module that defines a particular object type and all calls for objects of that type. This chapter shows you how to build a type manager.

Packages Used:

`Access_Mgt`	Interface for checking or changing rights.
`Object_Mgt`	Provides basic calls for objects.

The example for this chapter, `Account_Mgt_Ex`, is a simple, general-purpose type manager written as a Ada package. The complete listing of this example can be found in Appendix X-A.

VII-3.1 Concepts

A type manager provides both data abstraction and protection for the objects of its type. It does so by defining all calls for its objects. No operations but the ones defined by the type manager are possible on the objects protected by it. It is therefore important that you provide all necessary calls when building your type manager.

The type manager holds a key that allows it to create objects of its type and to add representation rights to ADs that are handed to it by calling programs. The key is an AD to the TDO with *amplify* and *create* rights. It is given out when the TDO is first created.

VII-3.1.1 The Type Manager Defines All Calls for a Type of Object

A type manager defines all basic calls for an object type. For example, the `Account_Mgt_Ex` type manager defines calls for *account* objects:

`Is_account`	Checks whether an AD references an account.
`Create_account`	Creates an account with an initial balance.
`Create_stored_account`	Creates and stores an account.
`Get_balance`	Returns an account's balance.
`Change_balance`	Changes an account's balance.
`Transfer`	Moves an amount between accounts.
`Destroy_account`	Destroys an account.

Callers must use the type manager `Account_Mgt_Ex` to do any of the above calls on an account. More complex calls must be composed from the type manager's basic calls. Again, it is important that the list of basic operations be complete, or else there is no way to do the operation on an account. For example, if you forgot the `Destroy_account` call, there would be no way to eliminate unneeded accounts.

VII-3.1.2 Type Managers Hide Data Representation

Type managers provide *data abstraction*, concealing the representation of data from callers. For example, `Account_Mgt_Ex` provides the calls `Create_account` and `Change_balance` that affect the data in an account. To other services, an account is an *abstract data type*; the caller doesn't need to know or care how data in the account is represented.

Data abstraction makes software more:

reliable — Only the type manager accesses the representation of a particular type of data. If the type manager is correct, then no outside program error can corrupt data of the type.

maintainable — Data representation can be changed as long as the correctness of the basic calls is preserved.

extensible — Changes in functionality can easily be implemented as long as they are compatible with the existing interface. In our example, operations on accounts could be realized using transactions without any other program but the type manager having to be changed.

VII-3.1.3 Only the Type Manager Has the Key to Access the Type's Objects

The type of an object is uniquely defined by the object's TDO. A TDO for a new type of object can be created with `Object_Mgt.Create_TDO`. `Object_Mgt.Create_TDO` returns an AD to the new TDO. This AD has *create* and *amplify* rights. Those are necessary to create new instances of the managed object, and to add access rights to ADs of managed objects. Any module that has a TDO with *create* rights and *amplify* rights is by definition a type manager for that type.

In order to protect a newly created type, the AD to the TDO that has *create* and *amplify* rights should be confined to your type manager.

VII-3.1.4 One Module Can Manage Multiple Types

The type manager model provides a flexible way of protecting objects. In particular, one module can manage as many types as you choose. However, it is obvious that the number of types that a type manager manages should be strongly limited. Otherwise the concept defeats itself. For example, it is common that one type manager manages closely related objects such as files and opened files.

VII-3.2 Techniques

This section shows you each step in building a type manager. After reading this section, you will be able to:

- Define the Public Type
- Define Type Rights
- Define Exceptions
- Define the Type's Calls
- Define the Private Types

- Define Needed Type Overlays
- Create the TDO
- Bind to a Stored TDO
- Implement the Is Call
- Implement the Create Call
- Implement Calls that Require Type Rights
- Implement Calls that Don't Require Type Rights
- Implement the Destroy Call
- Make Operations Atomic
- Initialize the Type Manager
- Protect the Type Manager from Other Services.

The first four techniques describe the type manager's package specification, the public interface used by outside callers.

The next eleven techniques describe the type manager's package body, the package implementation, which is hidden from outside callers.

The last technique describes how to use BiiN™ Ada pragmas and the BiiN™ Systems Linker to completely protect your type manager from other services.

The `Account_Mgt_Ex` example is a type manager for *accounts*, each containing a long integer balance. It is a general-purpose type manager and could be used for inventory accounts, bank accounts, or other accounting applications. Appendix X-A contains complete listings for the `Account_Mgt_Ex` package. Various implementations of this type manager are described in this chapter and in Chapters VII-6 and VIII-2. The implementation described in this chapter is the simplest and supports active-only accounts.

VII-3.2.1 Defining the Public Type

The type manager's package specification defines the *public type*, the type used by outside callers to reference an account. The `account_AD` access type is the public type for accounts. It references a private type `account_object` that is defined as a null record.

The package specification for `Account_Mgt_Ex` defines the public type:

```
type account_object is limited private;

type account_AD is access account_object;
  pragma access_kind(account_AD, AD);
  -- User view of an account.
```

The null record is defined in the private part of the specification:

```
private

  type account_object is
    -- Empty dummy record.  The real object
    -- format is defined in the package body.
    record
      null;
    end record;

end Account_Mgt_Ex;
```

A dummy record format is defined because the BiiN™ Ada compiler requires a record layout in the package specification, but it is still desirable to conceal the actual object representation in the package body. The `account_object` type is never actually used, because account ADs lack rep rights and cannot be used to read or write account objects. Actual reading and writing is done within the package body with types defined there.

VII-3.2.2 Defining Type Rights

Type rights allow a type manager to differentiate between users. The implementer of the type manager can require certain type rights for certain calls. It may also permit certain calls without any type rights. In the example presented here, the `Is_account` call is an example of a call that requires no type rights. (For more details, see Section VII-3.2.9.)

Declarations Used:

`Object_Mgt.rights_mask`
: Access rights type.

`Object_Mgt.modify_rights`
: Modify type right.

`Object_Mgt.control_rights`
: Control type right.

The type manager's package specification typically gives type-specific names to the type rights that it uses. The type manager's calls can check for needed rights before performing the call. A type manager does not always have to define all three rights. By convention, unused type rights should always be left turned on; otherwise a higher level routine will not be able to use them.

`Account_Mgt_Ex` defines two type rights:

```
change_rights:   constant
    Object_Mgt.rights_mask :=
    Object_Mgt.modify_rights;
  -- Required to change an account's balance.

destroy_rights:  constant
    Object_Mgt.rights_mask :=
    Object_Mgt.control_rights;
  -- Required to destroy an account.
```

If an account call is made without needed rights, then `System_Exceptions.insufficient_type_rights` is raised.

VII-3.2.3 Defining Exceptions

The type manager's package specification defines any type-specific exceptions raised by its calls. `Account_Mgt_Ex` defines these exceptions:

```
insufficient_balance:  exception;
  pragma exception_value(insufficient_balance,
      insufficient_balance_code'address);
  -- An operation failed because it would
  -- cause a negative account balance.

balance_not_zero:      exception;
  pragma exception_value(balance_not_zero,
      balance_not_zero_code'address);
  -- "Destroy_account" was called on an account
  -- with a nonzero balance.
```

Text messages to be displayed by CLEX when an exception occurs can be bound to these exceptions at compile-time. These messages can be displayed on a terminal, for example.

```
insufficient_balance_code:
    constant Incident_Defs.incident_code :=
    (0, 1, Incident_Defs.error, System.null_word);

--*D* manage.messages
--*D* store :module=0 :number=1 \
--*D* :msg_name=insufficient_balance_code \
--*D* :short= \
--*D* "An account operation failed because it\
--*D* would create a negative balance."

balance_not_zero_code:
    constant Incident_Defs.incident_code :=
    (0, 2, Incident_Defs.error, System.null_word);

--*D* store :module=0 :number=2 \
--*D* :short= \
--*D* "An account cannot be destroyed because\
--*D* it has a non-zero balance."
--*D* exit
```

VII-3.2.4 Defining the Type's Calls

The type manager's package specification defines all calls available to outside callers of the type.

Calls typically provided for a type *T* are:

`Is_T`	Checks whether an object is of type *T*. Only the type manager can reference *T*'s TDO and make this check.
`Create_T`	Creates a *T* object. Only the type manager can create and initialize *T* objects.
`xxx_T`	Any calls that need to read or write *T* objects. Only the type manager can read from or write to the object's representation.
`Destroy_T`	Destroys a *T* object. Only the type manager can explicitly deallocate *T* objects.

`Account_Mgt_Ex` defines all the typical calls:

`Is_account`

`Create_account`

`Create_stored_account`

`Get_balance`

`Change_balance`

`Transfer`

`Destroy_account`

It might appear at first glance that the `Transfer` call is not necessary since it can be composed of two calls to `Change_balance`. The problem with this solution is that it could happen that the calling program fails before it completes the transfer. Thus an amount may be deducted from the source account and not be deposited in the target account. The `Transfer` call is set up to be an atomic operation. It can only succeed as a unit and not partially. This concludes the type manager's package specification. The following techniques are done in the first body of `Account_Mgt_Ex`.

VII-3.2.5 Defining the Private Types

The type manager's package body defines the *private types* used inside the type manager to reference the accounts. The `account_rep_object` type defines the object's representation. The `account_rep_AD` type is used for ADs with rep rights, allowing the type manager to read and write the representation:

```
type account_rep_object is
  record
    balance:  Long_Integer_Defs.long_integer;
      -- Current balance.
  end record;

type account_rep_AD is access account_rep_object;
  pragma access_kind(account_rep_AD, AD);
  -- Private view of an account.
```

VII-3.2.6 Defining Needed BiiN™ Ada Type Overlays

The `Account_Mgt_Ex` package body requires three different BiiN™ Ada types to represent the AD to one of its objects:

`account_AD` Public AD without rep rights.

`System.untyped_word`
Type required for `Access_Mgt` and `Object_Mgt` calls.

`account_rep_AD`
Private AD with rep rights.

Instead of instantiating `unchecked_conversions` type overlays are used here to the same goal. This is done using a BiiN™ Ada *address clause*. (Refer to the *BiiN™ Ada Language Reference Manual* for more details.)

```
account_rep:  account_rep_AD;
FOR account_rep USE AT account'address;
account_untyped:  System.untyped_word;
FOR account_untyped USE AT account'address;
```

Note that this technique has no runtime cost.

VII-3.2.7 Creating the TDO

The package body described in this chapter is an *active-objects-only* package body, so every time the package initializes it creates a TDO. This poses no problems as long as objects of the type are not passivated or do not outlive their TDO or type manager. (This is explicitly enforced -- refer to Section VII-3.2.16 in this chapter for more details.)

```
account_TDO:  constant Object_Mgt.TDO_AD :=
                       Object_Mgt.Create_TDO;
```

A stored object should use a stored TDO as its type, as described in the next section.

VII-3.2.8 Binding to a Stored TDO

If objects of the type can outlive a particular job, then the TDO should be a stored object, created once by the system administrator.

The type manager's package body then uses the BiiN™ Ada `bind` pragma to obtain the needed TDO AD with all type rights. The following example is excerpted from the second body of `Account_Mgt_Ex` package body in Appendix X-A. In this example, the `account_TDO` is first assigned a null value, then used in the pragma `bind`:

```
account_TDO:  constant Object_Mgt.TDO_AD := null;
  -- This is a constant AD but not really null; its
  -- filled in with an AD retrieved by the linker.
  pragma bind(account_TDO,
              "account");
    -- Bind to TDO for accounts.
```

This technique declares a BiiN™ Ada access type variable which is initialized with `null` at compile-time. The BiiN™ Ada `pragma bind` is an instruction to the BiiN™ Systems Linker to retrieve an AD from the directory entry that is named by the second argument of `pragma bind`. (For more details on BiiN™ Ada pragmas refer to the *BiiN™ Ada Language Reference Manual.*) The linker reinitializes the variable with the activated AD.

VII-3.2.9 Implementing the `Is_account` Call

The `Is` call checks whether an object has the type managed by the type manager.

Calls Used:

`Object_Mgt.Retrieve_TDO`
Retrieves object's TDO.

`Is_account` returns true if `obj`'s type equals `account_TDO`, false if `obj` is null or has another type:

```
begin
  return obj /= System.null_word and then
         Object_Mgt.Retrieve_TDO(obj) = account_TDO;
end Is_account;
```

VII-3.2.10 Implementing the `Create_account` Call

The `Create` call allocates an object of the right size and type, initializes the representation, and returns an AD with no rep rights.

Calls Used:

`Object_Mgt.Allocate`
Allocates an object with specified size and type.

`Access_Mgt.Remove`
Removes rights.

The `Create_account` call creates an account with a specified `starting_balance`:

```
begin
  if starting_balance < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    account_untyped := Object_Mgt.Allocate(
        size => Object_Mgt.object_size(
                (account_rep_object'size + 31)/32),
          -- Expression computes number of words
          -- required to hold the number of bits
          -- in an account.
        tdo  => account_TDO);

    account_rep.all := account_rep_object'(
        balance => starting_balance);

    account_untyped := Access_Mgt.Remove(
        AD     => account_untyped,
        rights => Object_Mgt.read_write_rights);
    RETURN account;

  end if;
end Create_account;
```

The BiiN™ Ada `new` operator cannot be used here to allocate the object, because `new` by default allocates a generic object instead of an object with the desired type `account`. However, if we had made use of the Ada pragma `allocate_with` we could have specified a TDO to be used with the `new` operator. Thus we would obtain objects of the proper type when using `new`.

The `size` specified to `Allocate` is the number of 32-bit words. The BiiN™ Ada attribute `size` yields the number of bits required for the object's representation. The expression `(account_rep_object'size + 31)/32` yields the smallest number of 32-bit words with at least the required number of bits.

VII-3.2.11 Implementing the `Create_stored_account` Call

Our particular example provides two `Create` calls, one that simply creates an object and returns an AD, and another that also stores the object with a pathname. The implementation discussed in this chapter does not support stored objects, however. For this reason the the `Create_stored_account` function simply raises the `System_exception.operation_not_supported` exception as shown in the following excerpt from this implementation:

```
function Create_stored_account(
    starting_balance:
        Long_Integer_Defs.long_integer :=
        Long_Integer_Defs.zero;
    master:  System_Defs.text;
    authority:
        Authority_List_Mgt.authority_list_AD := null)
  return account_AD
  --
  -- Logic:
  --   This call is  not supported by this implementation.
  --
is
begin
  RAISE System_Exceptions.operation_not_supported;
  RETURN null;

end Create_stored_account;
```

VII-3.2.12 Implementing Calls that Require Type Rights

For calls that require type rights, the type manager checks the rights on the caller's AD before performing the requested operation. The usual way to do this is with `Access_Mgt.Import`, which checks type rights before adding rep rights. `Import` raises `System_Exceptions.insufficient_type_rights` if needed rights are not present.

Calls Used:

`Access_Mgt.Import`
: Checks for rights and adds rep rights.

Declarations Used:

`System_Exceptions.insufficient_type_rights`
: Raised when the AD does not have the type rights needed for the call.

In `Account_Mgt_Ex`, the call `Change_balance` requires that the caller have *change rights* on the passed AD:

```
begin
  account_untyped := Access_Mgt.Import(
      AD      => account_untyped,
      rights => change_rights,
      tdo     => account_TDO);

  new_balance := account_rep.balance + amount;

  if new_balance < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    begin
      old_balance := account_rep.balance;
      account_rep.balance := new_balance;
      RETURN new_balance;
    exception
      -- An exception in this inner block means
      -- that something has gone wrong with the
      -- update. The old balance is restored.
      when others =>
        account_rep.balance := old_balance;
      RAISE;
    end;

  end if;
end Change_balance;
```

The call `Access_Mgt.Import` checks the AD for *change rights* before adding rep rights.

VII-3.2.13 Implementing Calls that Do not Require Type Rights

Calls that don't require type rights don't need to check the type rights before performing the call. As a result, the type manager can use `Access_Mgt.Amplify`, which adds rights without doing a check for type rights.

Calls Used:

`Access_Mgt.Amplify`
Adds rights without checking type rights.

An example of a call that doesn't require type rights is `Account_Mgt.Get_balance`. In this case, read rep rights are amplified:

```
begin
  account_untyped := Access_Mgt.Amplify(
      AD      => account_untyped,
      rights => Object_Mgt.read_rights,
      tdo     => account_TDO);
  return account_rep.balance;
end Get_balance;
```

VII-3.2.14 Implementing the Destroy Call

A type manager's `Destroy` call usually checks type rights for this destructive act, then deallocates the object's representation.

Calls Used:

`Access_Mgt.Import`
Checks for rights and adds rep rights.

`Object_Mgt.Deallocate`
Deallocates the object's representation.

In the following example from `Account_Mgt_Ex`, the call `Object_Mgt.Import` checks for the appropriate type rights, then adds rep rights to the AD in order to be able to check the balance. If the balance in the account is zero, the account will be deallocated using `Object_Mgt.Deallocate`:

```
begin
  account_untyped :=  Access_Mgt.Import(
        AD     => account_untyped,
        rights => destroy_rights,
        tdo    => account_TDO);

  if account_rep.balance /= Long_Integer_Defs.zero then
    RAISE balance_not_zero;

  else
    Object_Mgt.Deallocate(account_untyped);

  end if;
end Destroy_account;
```

VII-3.2.15 Making Operations Atomic

Although the `transfer` call can in principle be composed of two successive calls to `Change_balance` there is a considerable disadvantage to this method; the process that performs the two calls could encounter an exception after performing the first call and before the second. If that happened, one account would be charged (or credited) but not the other one.

Calls Used:

`Access_Mgt.Import`
Checks for rights and adds rep rights.

```
begin
  source_untyped := Access_Mgt.Import(
      AD     => source_untyped,
      rights => change_rights,
      tdo    => account_TDO);
  dest_untyped := Access_Mgt.Import(
      AD     => dest_untyped,
      rights => change_rights,
      tdo    => account_TDO);

  new_source_bal := source_rep.balance - amount;
  new_dest_bal := dest_rep.balance + amount;

  if new_source_bal < Long_Integer_Defs.zero
     or else
     new_dest_bal < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    old_source_bal := source_rep.balance;
    old_dest_bal := dest_rep.balance;
    -- Old balances are recorded here
    -- in case the update will have to be
    -- rolled back.
    begin
      source_rep.balance := new_source_bal;
      dest_rep.balance := new_dest_bal;
    exception
      -- An exception in this inner block means
      -- that something has gone wrong with
      -- the update. Restore the old balances to make
      -- this operation atomic, then
      -- reraise the exception.
      when others =>
        source_rep.balance := old_source_bal;
        dest_rep.balance := old_dest_bal;
        RAISE;

    end;
    RETURN;

  end if;
end Transfer;
```

The new balances of both the source and the destination account are computed. If either one is less than zero, the `insufficient_balance` exception is raised. Before the balances in the accounts are physically changed, they are stored. Any exception that is raised while the new balances are assigned causes the update to be rolled back and the original balances to be restored.

VII-3.2.16 Initializing the Type Manager

The example that we discuss in this chapter manages accounts that cannot be passivated. In order to make sure that accounts cannot be passivated, the account TDO must contain the passive store attribute, bound to an instance that refuses requests for passive store operations.

Calls Used:

`Passive_Store_Mgt.Set_refuse_filters`
: Sets a type manager's passive store attributes object to refuse all outside requests for passive store operations.

`Attribute_Mgt.Store_attribute_for_type`
: Stores an attribute entry in a TDO.

```
begin
  Passive_Store_Mgt.Set_refuse_filters(
      passive_store_impl);
  Attribute_Mgt.Store_attribute_for_type(
      tdo       => account_TDO,
      attr_ID   => Passive_Store_Mgt.PSM_attributes_ID,
      attr_impl => passive_store_impl_untyped);
end;
```

Note that this piece of code is executed every time this package is initialized. Also, a new TDO is created at that time. The TDO and all the objects of the type manager are deallocated when the job that uses this package finishes.

A more general package body would be able to handle objects that can be passivated. In this case the TDO should only be created once and stored. This can be done by the system administrator using the `create.TDO` command in the `configure` utility. (For more details see the *BiiN™ Systems Administrator's Guide.*) You could also write a program that will execute only once, create a TDO and store it. The `Stored_Account_TDO_Init_Ex` procedure in Appendix X-A is an example of such a program.

VII-3.2.17 Protecting the Type Manager from Other Services

Finally, a type manager may want to protect its address space from other services so that it and its objects are safe from accidental destruction or modification. Protecting the type manager's address space involves:

1. Creating a distinct address space with the BiiN™ Systems Linker.
2. Protecting the type managers address space from calling services via pragma `protected_return`.

The idea is to link the type manager into its own separate domain. In addition it might be desirable to put the type manager into its own subsystem. That means that the type manager will not share stacks with other services.

Refer to the *BiiN™ Systems Linker Guide* for information on how to create the type manager's own address space at link time. You will need to create a distinct domain and a distinct subsystem ID.

The BiiN™ Ada pragma `protected_return` ensures that all global registers will be cleared before control is returned to the calling process. This is to protect ADs that may have been left in the global registers by the call. Refer to the *BiiN™ Systems Linker Guide* for more information on these topics. (Pragma `protected_call` is similar to `protected_return`; however it protects the calling routine from the routines it calls. `Account_Mgt_Ex` only calls OS routines. Therefore `protected_call` could be used here but is not really necessary.)

There is a performance penalty involved when you create a protected address space for a type manager. You will use extra memory for the type manager's distinct stack. There is also a time penalty when performing calls to a distinct domain.

VII-3.3 Summary

- A *type manager* defines an object type and all basic calls for the type.
- Only the type manager can read from or write to the type's objects.
- A type is represented by a TDO.
- Type managers provide *data abstraction*, enhancing software reliability and maintainability.

USING ATTRIBUTES 4

Contents

An *attribute* is a package or data structure that can be defined for multiple objects or object types. Such packages or structures can be used independent of an object's type and without calling its type manager.

An attribute usually defines a set of operations that is supported by multiple objects, or object types, such as an I/O access method.

Packages Used:

`Attribute_Mgt` Manages attribute IDs and provides calls to store and retrieve attribute instances.

`Object_Mgt` Provides basic calls on objects.

An attribute can be defined either for an object or for an object type. In case of type attributes, an attribute list is contained in the Type Definition Object (TDO). In the case of object attributes, an attribute list is attached to the object proper. Whether in the TDO or attached to an individual object, an attribute list contains one or more *<attribute ID, attribute instance>* pairs. The attribute ID in the pair identifies the attribute (for example, the Byte Stream Access Method). The attribute instance in the pair references the object- or type-specific attribute value (for example, the type-specific implementation of the access method for the particular device type). An example of an object-specific attribute is `execute`. An executable object can be a CLEX script, a BiiN™/UX script, or an executable program. The attribute instances in this case specify how the object is to be executed.

Figure VII-4-1 shows the attribute data structure for a type-specific attribute.

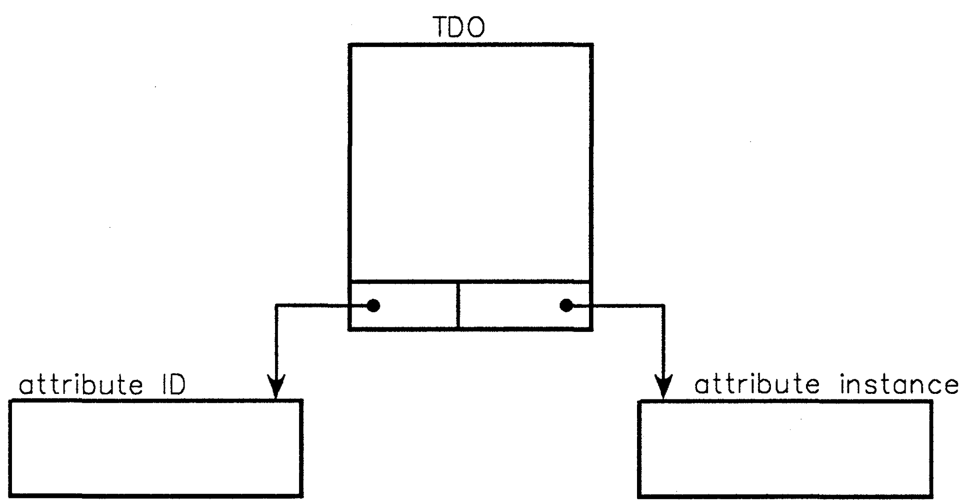

Figure VII-4-1. Attribute Structure

In this chapter you will find an example of how to use type-specific attributes. Using object-specific attributes is very similar to what is shown in the example. In addition, in each section you will find information on how to achieve the particular step for an object-specific attribute.

In a later release we may have an example of an object-specific attribute.

VII-4.1 Concepts

The attributes described in this chapter should not be confused with BiiN™ Ada attributes, used to indicate properties of declared entities in that language.

Even though *using* an attribute is independent of the object or its type, *defining* the attribute instances supported by an object or a type is specific to an object or a type. In the case of a type attribute, only the type manager can store attributes in the TDO, normally at system or program initialization when the TDO is created. In the case of an object attribute, anyone with control rights can store an attribute. But type-specific attributes cannot be overridden by object-specific attributes.

Though in most cases an attribute value is an AD to a package, an attribute value can be any `System.untyped_word`, either an AD to an object or a 32-bit data value. The attribute value can reference any object, not just a package. An example of an attribute value that does not reference a package is `Passive_Store_Mgt.PSM_attributes_object` where the attribute value is an AD to a record.

If an attribute is a package, invoking the attribute package's calls uses a fast *attribute call* mechanism supported by the OS and BiiN™ Ada. This mechanism uses the object type of the *first parameter* to a call to choose the appropriate type-specific instance of the package. This mechanism is used by many OS attributes, including all I/O access methods. If an attribute call is made on an object that does not support the attribute, then the `Standard.constraint_error` exception is raised. **The opinions vary on what exception will actually be raised. Also in the running are** `System_Exceptions.bad_parameter` **and** `System_Exceptions.operation_not_supported`.

Figure VII-4-2 shows an OS attribute, the Byte Stream Access Method, defined by the `Byte_Stream_AM` package, that is supported by different object types, such as opened files and opened pipes. Each object type has a type-specific implementation of the access method but applications need only call `Byte_Stream_AM` and their call is efficiently switched to the right implementation by the attribute call mechanism.

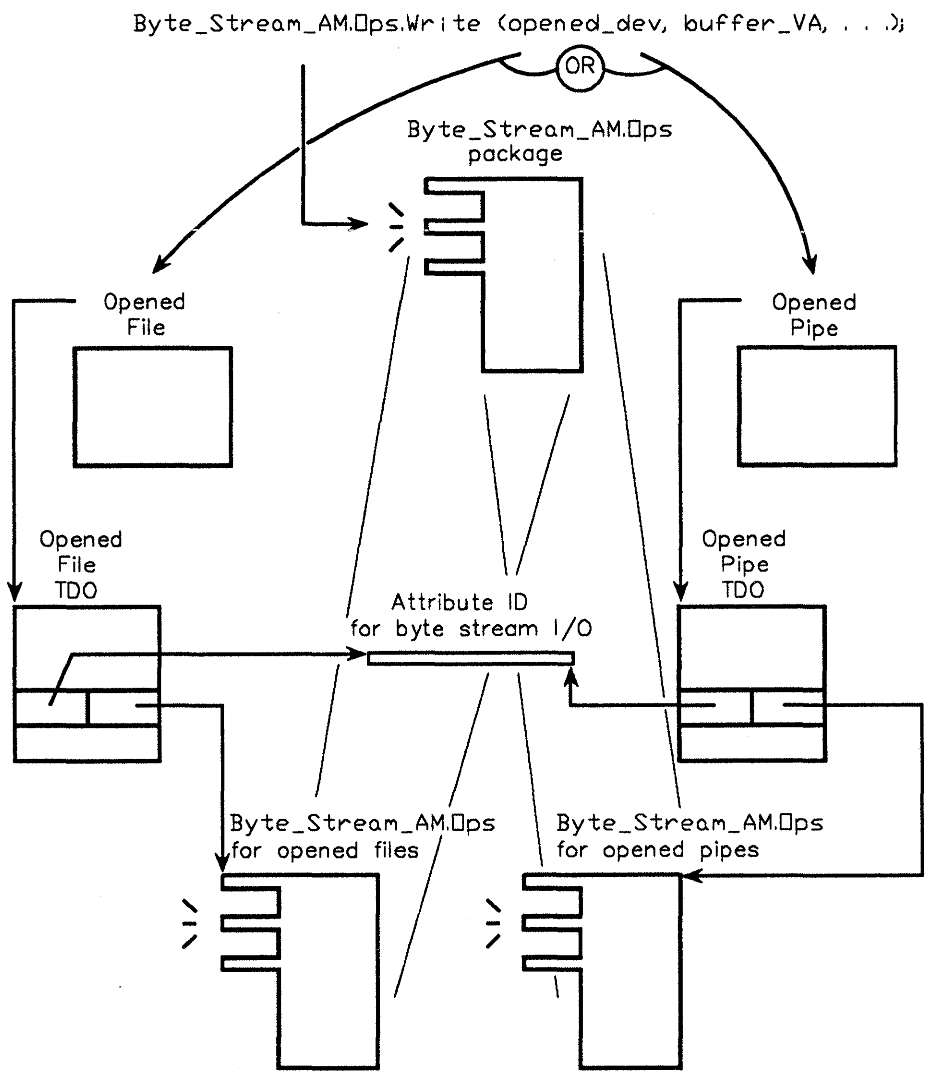

Figure VII-4-2. An OS Attribute

The OS defines many attributes used by type managers to customize System Services for their particular types. Every OS attribute appears to an application as another System Service. At the same time, implementers of new services can define type-specific instances of these OS attributes, without modifying, recompiling, or relinking the OS. *You can use attributes to extend and customize the OS -- without accessing its internals in any way.*

The "OS Attributes" appendix in the *BiiN™/OS Reference Manual* summarizes all OS attributes. Some commonly used OS attributes are:

- Byte stream I/O, specified by the `Byte_Stream_AM.Ops` package.
- Record I/O and record keyed I/O, specified by the `Record_AM.Ops` and `Record_AM.Keyed_Ops` packages.
- Character display I/O, specified by the `Character_Display_AM.Ops` package.

- Passive store, specified by the `Passive_Store_Mgt.PSM_attributes_object` record type.
- The execute attribute, specified by `Execution_Support.Ops`, an example of an attribute that can be object-specific.

VII-4.2 Techniques

There are three techniques in using attributes:

- Defining a new attribute
- Defining a type-specific attribute instance for a type
- Initializing the type's TDO to refer to the attribute and instance.

Because attributes are most often packages, this section uses a simple package attribute for all three examples. This attribute contains a single call, which returns a type-specific type name. For example, for account objects, the type-specific instance will return the string `"account"`. This example is not as useful as many attributes, such as I/O access methods, but its simplicity allows you to easily understand programming with attributes.

VII-4.2.1 Defining a New Attribute

You will more often define attribute instances than define new attributes. We begin with defining an attribute because the example attribute is used by the subsequent techniques.

Calls Used:

`Attribute_Mgt.Create_attribute_ID`
Creates a new attribute ID.

You create a new attribute by calling `Attribute_Mgt.Create_attribute_ID`. In this call you can specify whether the new attribute is type-specific or not. Type-specific attributes can only be stored in a TDO and not in an object's attribute list. The newly created attribute ID should be stored in the `aid` directory in the node's root directory.

The `Type_Name_Attribute_Ex` example package assumes that the attribute has already been created and stored. It binds the previously created ID to an attribute package using the BiiN™ Ada pragma `bind`.

```
type_name_attr_ID:  constant
    Attribute_Mgt.attribute_ID_AD := null;
  pragma bind(type_name_attr_ID,
              "typnamattr");
  -- Attribute ID is retrieved at link time using the
  -- specified pathname.  Should have store rights.
```

The attribute package `Type_Name_Attribute_Ex` defines two functions: one to get the attribute ID and one to return a type's name.

The `Get_type_name_attr_ID` function returns the new attribute's ID, required to store an instance of the type-name attribute:

```
function Get_type_name_attr_ID
  return Attribute_Mgt.attribute_ID_AD;
    -- Type name attribute ID, with type rights.
  --
  -- Function:
  --   Returns the type name attribute's attribute ID.
```

The nested `Ops` package contains the calls to be defined by each type-specific instance. Only subprograms can be declared in such a package. The `package_type` pragma declares the nested `Ops` package to be a package type.

```
package Ops is
  pragma package_type("typnamattr");
    --
    -- Function:
    --   Provide "Type_name" attribute call.
  function Type_name(
      obj:  System.untyped_word)
        -- Any object that supports
        -- the type name attribute.
    return string;  -- Name of the object's type.
    pragma interface(value, Type_name);
    --
    -- Function:
    --   Returns a printable name for an object's type.
end Ops;
```

Calls to any operations declared in the `Ops` package are switched to the proper instance, using the the *first* parameter to the call to select the instance.

The `Ops.Type_name` function body is empty. An empty subroutine body is allowed here due to the `package_type` pragma:

```
package body Ops is
  --
  -- Logic:
  --   Attribute packages have null bodies.
end Ops;
```

Defining the attribute is done no differently for an object-specific attribute. In fact, an attribute that is not labeled as type-specific can be added to the attribute list of an object.

VII-4.2.2 Defining an Attribute Instance

An attribute instance is simply a package that matches ("conforms to") the attribute's `Ops` package template and that is bound to that template using the `package_value` pragma:

```
with System,
     Type_Name_Attribute_Ex;

package Account_Type_Name_Ex is
  pragma package_value(Type_Name_Attribute_Ex.Ops);
  --
  -- Function:
  --   Defines the type name attribute for accounts.
  --
  --   A type that supports this attribute has a
  --   printable name.  For example, a directory
  --   listing utility could use this attribute to
  --   print the types of the objects in a
  --   directory.


  function Type_name(
      obj:  System.untyped_word)
    return string;
      -- Name of the "account" object type.
    --
    -- Function:
    --   Returns the type name for account objects.


  pragma external;

end Account_Type_Name_Ex;
```

Note that the instance does not contain a nested `Ops` package. It corresponds to the attribute's nested `Ops` package and it will be called whenever one of the general `Ops` routines is called with a first parameter that is an object to which the attribute applies. Note that pragmas `package_value` and `package_type` occur paired. They can be compared to a *type definition* and a *variable declaration* in BiiN™ Ada.

The `Account_Type_Name` package body simply returns the name `"account"` when its `Type_name` function is called:

```
with System;

package body Account_Type_Name_Ex is


  function Type_name(
      obj:  System.untyped_word)
    return string
  is
  begin
    return "account";
  end Type_name;


end Account_Type_Name_Ex;
```

VII-4.2.3 Initializing the Type's TDO

Calls Used:

`Attribute_Mgt.Store_attribute_for_type`
Stores attribute ID and instance in TDO.

The implementation of the type-name attribute for accounts must be stored in the account TDO to be useful. The following excerpt is from the `Stored_Account_Init_Ex` example package body:

```
type_name_impl:  System.untyped_word;
  -- Implementation of type name attribute
  -- for accounts.
...
type_name_impl := Account_Type_Name_Ex'package_value;

Attribute_Mgt.Store_attribute_for_type(
    tdo       => account_TDO,
    attr_ID   => Type_Name_Attribute_Ex.
                 Get_type_name_attr_ID,
    attr_impl => type_name_impl);
```

The `'package_value` BiiN™ Ada attribute (not to be confused with an OS attribute) is used to obtain an AD for the type-specific `Account_Type_Name_Ex` package, an AD which is then stored in the TDO.

Handling TDOs and attributes that are stored objects is described in Chapter II-3.

VII-4.2.4 Initializing an Objects Attribute List

Calls Used:

`Attribute_Mgt.Retrieve_attribute_list`
Get's an object's attribute list. If none exists, creates one.

`Attribute_Mgt.Store_attribute_for_object`
Stores attribute ID and instance in TDO.

Before you can use an object-specific attribute you have to store it in the object's attribute list. To do so, ou have to retrieve the attribute list with `Attribute_Mgt.Retrieve_attribute_list`. This returns an AD to the object's attribute list. If none exists, a new attribute list is created. Finally, you can store the attribute using `Attribute_Mgt.Store_attribute_for_object`.

VII-4.3 Summary

- An *attribute* is a package or data structure that can be defined for multiple objects or types.
- Explicitly type-specific attributes can only be associated with a type, not any object.
- An attribute *instance* is an attribute's value for a particular object or type.
- Attributes are identified by *attribute ID* objects.
- A type manager stores type_specific attribute instances of attributes that it supports in its TDO.
- Anyone with control rights to an object and store rights to an attribute can store that attribute in the object's attribute list.

MANAGING ACTIVE MEMORY 5

Contents

This chapter points out how you can use certain tools to manage active memory. This chapter does not explain underlying concepts and models of memory management in a BiiN™ system. Refer to Chapter VII-2 for a conceptual explanation of active memory.

For the most part, memory is managed automatically by the OS. You will want to read this chapter if you want to use optional calls to monitor and control your program's memory use.

Packages Used:

`Object_Mgt`	Provides basic calls on objects. Includes a call to shrink the calling process's stack.
`SRO_Mgt`	Provides calls to get memory information and control local garbage collection.

VII-5.1 A Brief Overview of How Memory Is Allocated

Virtual address space in active memory is managed on a per-job and per-node basis. Each job has a special type of object associated with it that represents memory and objects local to the job and shared by all its processes. This object is known as a *local storage resource object* (SRO).

A local SRO provides a job with its own local address space, a subset of the node's virtual address space. Objects in the address space can be reclaimed by starting a local garbage collection daemon. The daemon is basically a memory optimization technique used for long-running jobs. It deallocates unreferenced objects (that is, objects with no ADs). See the `SRO_Mgt.Start_GCOL` call.

NOTE

Local garbage collection should be started in long-running jobs that need to respond quickly to events, terminal input, or other stimuli. If local garbage collection is not started by the job itself, then local garbage collection is done synchronously whenever the job reaches one of its memory limits. Synchronous local GCOL suspends all other processes in a job until it completes.

NOTE

Memory resources can be consumed by system calls other than those that explicitly allocate memory. For example, every time a transaction is started, the transaction counts against the job's "countable object" limit, even after the transaction is committed or aborted. Local GCOL will detect that the job is not using the transaction any longer and will decrement the job's "countable object count" accordingly.

Some more information about the local SRO:

- The local SRO is shared by all processes in the job, and only by the processes in the job.
- All processes in a job have implicit access to their job's local SRO.
- Most object allocation operations require an SRO as a parameter. This parameter defaults to the local SRO of the job to which the calling process belongs.

SROs have a number of properties that indicate how the objects allocated from an SRO are treated by various memory management functions. These properties are:

relative lifetime Determines when objects can be *deleted* (that is, deallocation of both the object's representation and its unique object descriptor) and constrains the storing of ADs in objects.

memory type Determines whether or not parts of an address space can be relocated.

memory priority Determines the frequency with which unused pages are swapped out of active memory; also determines when small segments are compacted onto a single page.

allocation limits Determines the amount of virtual storage allowed for all objects allocated.

Each one of these properties is discussed in more detail in Chapter VII-2.

VII-5.2 Collecting Garbage Objects -- GCOL

Unreferenced objects in active memory (that is, objects with no active ADs) are periodically collected and deleted. This garbage collection (GCOL) is generally done automatically by the system, although it can be configured to clean up local objects for long-running jobs.

VII-5.2.1 Local GCOL

Local garbage collection is executed by a special daemon process in a particular job. The daemon is *only* present if a process in the running job requests it and can be deleted at times when no garbage collection is needed.

It is useful to configure local GCOL for long-running jobs. When local garbage collection is configured for a job, it can be triggered in one of two ways:

- Automatically, whenever one of the remaining claim values becomes smaller than a percentage of the original claim set by the programmer.
- Manually, by calling `SRO_Mgt.Start_GCOL` with all parameters defaulted.

The effect of a `SRO_Mgt.Start_GCOL` depends on the values of the parameters. Table VII-5-1 summarizes the key parameters. Selected parameter combinations are used to start the daemon manually and then to stop GCOL by deleting the daemon. See "Techniques" in this chapter.

Table VII-5-1. Key GCOL Parameters

Parameter	Description
`storage_claim_percent`	Threshold value at which GCOL daemon wakes up. A percentage of the original number of words of virtual space that the specified SRO is allowed to allocate.
`OTP_claim_percent`	Threshold value at which GCOL daemon wakes up. A percentage of the original number of object table pages (OTP) assigned for the specified SRO.
`minimum_delay`	Minimum time between runs of the GCOL daemon.

This can have the effect of starting up the daemon. To prevent the daemon from running too often, a *minimum delay* can be specified as one of the trigger parameters. Garbage collection will not be triggered automatically if the elapsed time since it started its previous run is smaller than the minimum delay. Table VII-5-2 lists the special parameter values and their effect.

Table VII-5-2. GCOL Parameters to Start and Stop Special GCOL

Effect	Stop GCOL	Start GCOL
storage_claim_percent	0	100
OTP_claim_percent	0	100
minimum_delay	max_int	null_time

The max_int and null_time constants are defined in the Long_Integer_Defs and System_Defs packages under "Support Services."

The garbage collection algorithm has these properties:

- Only objects that are garbage at the time the algorithm starts will be collected.
- Garbage objects are deleted during the final phase of the algorithm.

SRO_Mgt.Read_SRO_information returns garbage collection related information.

Figure VII-5-1 shows the algorithm used by the system to determine when global garbage collection is performed:

```
          % remaining_storage_claim < storage_claim_percent
         /
      OR
     /  \\
    /     % remaining_OTP_claim < OTP_claim_percent
AND
    \\
      start_time + minimum_delay < current_time
```

Figure VII-5-1. Algorithm That Controls Garbage Collection

SRO_Mgt.Start_GCOL parameters specify when the GCOL daemon should begin running. When either of the claims granted to the job's local SRO drops below the trigger values and the minimum delay condition is met, the daemon starts running.

VII-5.2.2 Global GCOL

Global garbage collection runs periodically and collects garbage objects allocated from both global SROs. Since global ADs may be stored in any object, all objects (local *and* global) on the node are checked. As with local garbage collection, objects and their associated space are only deleted during the final phase of the algorithm. Internally, the system minimizes the need for global garbage collection by minimizing the generation of global garbage.

VII-5.3 Techniques

After reading this section, you will be able to:

- Trim the caller's stack
- Start local garbage collection
- Stop local garbage collection
- Get information about a job's local memory.

All techniques are taken from the `Memory_ex` example in Appendix X-A.

VII-5.3.1 Trimming the Caller's Stack

A process can use an event handler to trim its stack in response to the `Event_Mgt.gcol` local event which is signalled to each process in a job whenever a local GCOL daemon is triggered.

Calls Used: `Object_Mgt.Trim_stack`
Shrinks the calling process's stack.

Basically, `Trim_stack` looks at the process's current call stack pointer and then resizes the stack.

```
29    Object_Mgt.Trim_stack;
```

Trimming the stack frees memory and reduces the number of ADs that the local GCOL daemon must scan, thus speeding up garbage collection.

VII-5.3.2 Starting Local Garbage Collection

To trigger local GCOL to start immediately in the calling job, you can use default parameters.

Calls Used:

`SRO_Mgt.Start_GCOL`
Controls the local GCOL daemon.

For example:

```
35    SRO_Mgt.Start_GCOL;
```

This will trigger the GCOL daemon to begin reclaiming space allocated from the job's local SRO.

VII-5.3.3 Setting/Changing Local GCOL Parameters

Local GCOL parameters can be configured to trigger the local GCOL daemon. The daemon is triggered only when the conditions specified in the configuration are met.

Calls Used:

`SRO_Mgt.Start_GCOL`
Controls the local GCOL daemon.

For example, you might want to configure a local garbage collection daemon to run in the calling job when it has used 50% of its storage claim *or* 50% of its object table page claim, *and* at least 5 minutes has elapsed since a previous local GCOL run in the job.

```
SRO_Mgt.Start_GCOL(
    storage_claim_percent => 50,
    OTP_claim_percent     => 50,
    minimum_delay         =>
        Long_Integer_Defs."*"(
        Long_Integer_Defs.long_integer'(0, 5),
        System_Defs.stu_per_min));
```

VII-5.3.4 Stopping Local Garbage Collection

A local GCOL daemon, once started, can be stopped using a `Start_GCOL` call.

Calls Used:

`SRO_Mgt.Start_GCOL`
Controls local GCOL.

For example:

```
SRO_Mgt.Start_GCOL(0, 0, Long_Integer_Defs.max_int);
```

This will kill any local garbage collection daemon in the calling job. It does nothing if there is no daemon.

VII-5.3.5 Getting Information About a Job's Local Memory

To obtain information about the current status of a job's local memory, call `SRO_Mgt.Read_SRO_information`.

VII-5.4 Summary

- Active memory consists of primary memory and swap space.
- A node's active memory contains objects used by executing programs.
- A one-to-one mapping exists between local SROs and jobs.
- Most active objects are allocated from local SROs.
- Global memory is allocated from global SROs.
- There are two types of global SROs: frozen global SROs and normal global SROs that indicate whether reclamation and compaction is allowed in global memory.
- Garbage collection can be configured for objects allocated from local SROs; it has certain trigger values that initiate a daemon process used to reclaim space.

BUILDING TYPE MANAGERS FOR STORED OBJECTS 6

Contents

This chapter describes how to build a type manager for stored objects. The type manager has the following characteristics:

- Objects can be passivated.
- Transactions ensure the consistency of passive versions.
- The multiple activation model is used.
- Objects should not be used by concurrent processes in one job.

The techniques necessary are illustrated by way of an implementation of the `Account_Mgt_Ex` example introduced in Chapter VII-3. The example used in this chapter has an interface identical to the one previously discussed. This is reflected by the fact that the Ada specification is identical for both packages. In addition to the packages described here, there is another implementation of `Account_Mgt_Ex` provided in Appendix X-A. That implementation is slightly simpler and does not provide transaction-oriented calls. The transaction-oriented implementation for stored accounts will be referred to simply as the implementation of `Account_Mgt_Ex`. If any other implementation is referred to, that fact will be explicitly stated. (All example packages used in this chapter can be found in in Appendix X-A.)

This chapter is self-contained. It explains all techniques necessary for building a type manager for stored objects. It does not, however, discuss the fundamentals of the type manager model. If you do not know or understand the type manager model of protection, please read Chapters VII-1 and VII-3 before reading this chapter.

VII-6.1 Concepts

Active memory is the immediate working space of the processors in one node. Active memory is (relatively) small, volatile, and local to a node. Passive store is not limited in size, permanent, and global to a distributed system. Objects that should survive shutdowns or system crashes, or that should pass between node boundaries, have to be passivated. A type manager that stores its objects is distributed by virtue of the distributed nature of passive store.

VII-6.1.1 Storing and Retrieving Objects in Passive Store

All objects are created as active objects. *Local* active objects disappear when the creating job finishes. *Global* active objects survive as long as the system is up. Objects have to be passivated explicitly. Objects that have been passivated pass transparently between passive store and active memory.

Objects can be labeled *active-only*. Active-only objects cannot be passivated.

A job retrieves a stored object either transparently by supplying an AD or explicitly through a directory pathname. A job can also explicitly request that its current active version be updated from the passive version.

To remove an object that has been passivated, both the active version and the passive version have to be removed. Passive versions have always to be removed explicitly. Deallocating an object's active version has no effect on any existing passive version.

VII-6.1.1.1 Lifetime Requirements

Objects have a type defined by a *Type Definition Object* (TDO). The TDO acts as a label for the type and it holds information specific to the type. An object may also have an *attribute list.* The lifetimes of TDO and attribute list should be at least as long as the object's own lifetime. For this reason TDO and attribute list have to be passivated before any object is passivated.

An object that has not explicitly been assigned a TDO or whose TDO has been removed is assigned the *generic TDO* by default. This may have certain undesirable consequences. For more details refer to Section VII-6.1.2.

VII-6.1.1.2 Storing Objects Requires Three Steps

Storing an object for the first time requires three steps:

- TDO and attribute list is stored. If the TDO already exists this step is omitted.
- An AD is stored on the volume set where the object is to be stored. This AD can be stored in a directory or in another object. It will become the stored object's master AD. Master ADs cannot reference across volume sets.
- The object's representation is stored.

Once an object has a passive version, only its representation has to be updated if changes to the active version have been made. Note, that changes to an active version do not become permanent until the passive version has been updated.

VII-6.1.1.3 Object Trees in Passive Store

Master ADs can be stored inside other objects. Thus hierarchical trees of passive objects can be created where one object holds master ADs for objects one level below. Object trees can be copied, and updated as one unit. Activating the root object of an object tree does not activate all the objects in the tree. Only the root object will be activated and all its ADs converted from passive to active form.

VII-6.1.2 The Type Manager Can Customize Passive Store Operations

A type manager can supply its own routines for certain passive store operations thus customizing passive store. The mechanism behind this feature is an *attribute call.* For more details on attribute calls, refer to Chapter VII-4.

Passive store provides pairs of calls, *operation* and `Request_`*operation* calls. Direct calls, such as `Update`, require representation rights, while `Request_`*operation* calls, such as `Request_Update`, generally require only type rights. One exception are generic objects which require `read` representation rights for `Request_operation` calls. (The BiiN™ Operating System acts as a type manager for these objects.)

If upon invoking any `Request_`*operation* call you receive the `System_Exceptions.insufficient_rep_rights` exception, this is an indication that something has gone wrong with your TDO. It probably means that either the TDO could not be retrieved because you had insufficient rights to it or that it has been deleted altogether. Remember though that the type manager has total control over what actually happens when `Request_`*operation* is called. (The type manager could conceivably require `rep` rights for these operations.)

If a type manager does not exlicitly provide an implementation for a `Request_`*operation* call, the call is mapped by passive store to the direct call. This makes the direct call accessible with only type rights. Therefore, if any particular passive store operation should be disabled, an implementation of the corresponding `Request_`*operation* operation that refuses the operation, by raising an exception, for example, has to be provided. Otherwise the operation will be available to anyone with type rights.

VII-6.1.3 Synchronizing Access to Objects -- Transactions and Semaphores

The use of transactions in passive store operations ensures that the stored data is consistent even in the event of system failures. Transactions also coordinate between different jobs accessing an object in passive store. Passive store operations either participate in a caller's default transaction, or a transaction is started for the duration of the call to passive store. Transactions have a built-in blocking protocol that avoids circular blocking of transactions.

Semaphores coordinate access to active objects, typically between processes inside one job. If in the object layout a *locking area* has been provided, passive store transparently creates a semaphore upon activation. A process can also explicitly create a semaphore. This is necessary if the object has never been passivated or is active-only. Semaphore locking is not used in the example described in this chapter. For more details on semaphore locking refer to Chapters VI-1, VI-2, and VIII-1.

It is important to note the conceptual difference between transaction locking and semaphore locking. Transaction locking directly locks an object. While a transaction holds its lock it blocks all others that request access. Sempahore locking relies on voluntary compliance by all participating processes. Semaphore locking is therefore used primarily to coordinate between related processes, for example inside one job.

VII-6.2 Techniques

Packages Used:

`Access_Mgt` Interface for checking and changing rights in access descriptors.

`Attribute_Mgt` Provides a way to define general-purpose operations supported by multiple object types or objects, with different type-specific or object-specific implementations.

`Authority_List_Mgt`
Provides Calls to manage authority lists and to evaluate a caller's access rights to objects protected by authority lists.

`Directory_Mgt` Manages directories and directory entries.

`Identification_Mgt`
Provides operations to manage IDs and ID lists.

`Object_Mgt` Provides basic calls for object allocation, typing, and storage management. Defines access rights in ADs.

`Passive_Store_Mgt`
Provides a distributed object filing system.

`Transaction_Mgt`
Provides *transactions* used to group a series of related changes to objects so that either all the changes succeed or all are rolled back.

`User_Mgt` Provides calls to manage a user's protection set and user profile.

This section describes the techniques necessary for a complete implementation of a type manager. The example described in this chapter and the example described in Chapter VII-3 share the same specification. Therefore, please refer to Chapter VII-3 for the following techniques:

- Defining the public type
- Defining type rights
- Defining exceptions
- Defining the private types
- Binding to a stored TDO.

VII-6.2.1 Defining the Type's Calls

The implementation described in this chapter provides the same calls as the one discussed in Chapter VII-3. Some calls work a little differently, though:

`Is_account` Checks whether an AD references an account.

`Create_account`
Creates an account. Caller is responsible for storing the account.

`Create_stored_account`
Creates and stores an account. Caller supplies a pathname that is not already in use.

`Get_balance` Returns an account's current balance.

`Change_balance`
Adds or substracts an amount from the account's current balance.

Transfer — Transfers amounts between accounts. Transfer either completes or fails as a unit.

Destroy_account — Removes an account's active and passive versions. May leave a master AD behind.

The implementation of the Is_*type* call will not be discussed here as it is identical to the one discussed in Chapter VII-3. For details, refer to that chapter.

VII-6.2.2 Implementing the Create_account call

The Create_account call allocates an object of the right size and type, initializes the representation and returns an AD with no rep rights.

Calls Used:

Object_Mgt.Allocate — Allocates an object of specified size and type.

Object_Mgt.Deallocate — Removes an objects active version.

Access_Mgt.Remove — Removes rights on an AD.

The following excerpt from the implementation of Account_Mgt_Ex shows all the steps in the Create_account call:

```
begin
  -- 1. Check the initial balance:
  --
  if starting_balance < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    -- 2. Allocate and initialize the account object:
    --
    account_rep_untyped := Object_Mgt.Allocate(
        size => (account_rep_object'size + 31)/32,
        tdo  => account_TDO);
    begin
      -- Inside this block it is guaranteed
      -- that the object has been allocated.
      account_rep.all := account_rep_object'(
          balance => starting_balance);

      -- 3. Remove rep rights for the exported AD:
      --
      account_untyped := Access_Mgt.Remove(
          AD     => account_rep_untyped,
          rights => Object_Mgt.read_write_rights);

    exception
      -- 4. If any exception occurs, abort any local
      --    transaction, deallocate the account,
      --    and reraise the exception:
      --
      when others =>
        Object_Mgt.Deallocate(account_untyped);
        RAISE;

    end;

    RETURN account;

  end if;
end Create_account;
```

`Object_Mgt.Allocate` is used to allocate an object of the right size and type. This call can be substituted by the Ada `new` function if the BiiN™ Ada `allocate_with` pragma is specified with the private object type.

As can be seen from the above example, the `Create_`*object* call does not passivate the new object. It is the caller's responsibility to store the object. Note also, that if an exception occurs during the call after the account has been allocated, it will be deallocated and the exception reraised.

VII-6.2.3 Implementing the `Create_stored_account` Call

The `Create_stored_account` call allocates an object of the right size and type, stores a master AD under a pathname provided by the caller, updates the passive version, and returns an AD with all type rights and no rep rights. This call illustrates all steps necessary in storing an object. In addition, you will learn how to employ transactions to protect passive store operations.

Calls Used:

`Object_Mgt.Allocate`
Allocates an object of the right type and size.

`Access_Mgt.Remove`
Removes rights.

`Transaction_Mgt.Get_default_transaction`
Gets the caller's default transaction.

`Transaction_Mgt.Start_transaction`
Starts a local transaction.

`Transaction_Mgt.Abort_transaction`
Aborts a transaction. Rolls back any changes done by transaction oriented calls within the transaction.

`Transaction_Mgt.Commit_transaction`
Commits a transaction. Finalizes changes made within the transaction.

`Directory_Mgt.Store`
Stores an AD with a pathname.

`Passive_Store_Mgt.Update`
Updates a passive version.

The `Create_stored_account` call allocates an object and removes rights on the exported AD the same way the `Create_account` call does.

VII-6.2.3.1 Starting, Commiting, and Aborting a Transaction

All passive store operations in this call are enclosed in a transaction, either a caller's default transaction, or a local transaction. The following excerpt from the implementation of `Account_Mgt_Ex` illustrates the use of a local transaction.

```
-- 4. Start a local transaction if there is not
--    a transaction on the stack:
--
if Transaction_Mgt.Get_default_transaction =
    null then
  Transaction_Mgt.Start_transaction;
  trans := true;
end if;
begin
. . .
  if trans then
    Transaction_Mgt.Commit_transaction;
  end if;
exception
  -- 8. If any exception occurs, abort any local
  --    transaction, deallocate the account,
  --    and reraise the exception:
  --
  when others =>
    if trans then
      Transaction_Mgt.Abort_transaction;
    end if;
    Object_Mgt.Deallocate(account_untyped);
    RAISE;

end;
```

This technique avoids starting a local transaction if the caller already supplied a default transaction. Subtransactions should be avoided, unless specifically needed.

The above example also indicates the use of a program block to control the scope of the exception handler. Within this block one can assume that, if `trans` is true, a local transaction has indeed been started.

VII-6.2.3.2 Storing the Master AD

The next step in storing the object is to store the master AD. The following excerpt from the implementation illustrates the call to `Directory_Mgt`.

```
        Directory_Mgt.Store(
            name    => master,
            object  => account_untyped,
            aut     => authority);
```

`master` is a text record that contains the pathname to store the account. The pathname must reference an existing directory and not be in use. If the caller did not specify an authority list, `authority` is null, and the target directory's default authority list will be used, if one exists. Otherwise the caller's default authority list will be used. If no default authority list is found, the exception `Directory_Mgt.no_default_authority_list` is raised.

VII-6.2.3.3 Updating the Object

In the last step the object's representation is stored by calling `Passive_Store_Mgt.Update`:

```
        Passive_Store_Mgt.Update(account_rep_untyped);
```

Note, that storing the AD does not passivate the object's representation. If you omit this last step, a later attempt to retrieve the object will result in the `System_Exceptions.object_has_no_representation` exception being raised.

VII-6.2.4 Implementing the `Change_balance` Call

This call is a typical example of a type-specific operation. It illustrates the use of transactions to coordinate access to the passive version of an object between different jobs.

Calls Used:

`Access_Mgt.Import`
: Checks and amplifies rights on an AD in one step.

`Transaction_Mgt.Get_default_transaction`
: Returns the caller's default transaction.

`Transaction_Mgt.Start_transaction`
: Starts a local transaction.

`Transaction_Mgt.Abort_transaction`
: Aborts a transaction.

`Transaction_Mgt.Commit_transaction`
: Commits a transaction.

`Passive_Store_Mgt.Reserve`
: Reserves a passive version of an object on behalf of a transaction.

`Passive_Store_Mgt.Update`
: Updates an object's passive version.

Two steps are necessary before any operations can be performed on the object; the type rights have to be checked on the AD supplied by the caller, and representation rights have to be amplified. The following excerpt from the implementation illustrates the `Access_Mgt.Import` call that performs these two steps together:

```
account_untyped := Access_Mgt.Import(
        AD     => account_untyped,
        rights => change_rights,
        tdo    => account_TDO);
```

If the AD's type rights are insufficient, this call will result in the `System_Exceptions.insufficient_type_rights` exception being raised.

Before checking for a sufficient balance in the account, the technique described in the previous section is used to ensure that there is a default transaction. Next, the call reserves the passive version on behalf of the transaction:

```
    Passive_Store_Mgt.Reserve(account_untyped);
```

The `Passive_Store_Mgt.Reserve` call may have three different outcomes:

- The object is available. The call succeeds and locks the object on behalf of the default transaction.
- The object is locked by another transaction. The blocking protocol permits blocking. The call blocks until the object becomes available.
- The object is locked by another transaction. The blocking protocol does not allow blocking. The call returns with the `System_exceptions.transaction_timestamp_conflict` exception.

You have to be prepared to handle this exception. The technique used here is illustrated by the following excerpt from the implementation:

```
loop
  if Transaction_Mgt.Get_default_transaction =
      null then
    Transaction_Mgt.Start_transaction;
    trans := true;
  end if;
. . .
  exception
    when System_Exceptions.
         transaction_timestamp_conflict =>
      if trans then
        Transaction_Mgt.Abort_transaction;
      else
        RAISE;
. . .
  end;
end loop;
```

The `Passive_Store_Mgt.Reserve` operation is enclosed in a program block that has an exception handler for the `transaction_timestamp_conflict` exception. The block in turn is enclosed in a loop that repeats the `Reserve` call until it succeed in either blocking or reserving the object.

You can avoid the `Reserve` call. In that case, if the object had been updated by another job while your call was holding it, passive store would raise the `Passive_Store_Mgt.outdated_object_version` exception. You would handle the exception, request a fresh active version, by calling `Passive_Store_Mgt.Reset_active_version`, redo the changes, and try another update. This technique is not acceptable for our example, since it might result in the decision, whether the balance be changed, being based on an outdated balance.

VII-6.2.5 Implementing the `Transfer` Call

The `Transfer` call is similar in nature to other type-specific calls. It is discussed in more detail here, since it gives another example of how transactions can be used to keep data in passive store consistent.

Calls Used:

`Access_Mgt.Import`
: Checks and amplifies rights on an AD in one step.

`Transaction_Mgt.Get_default_transaction`
: Returns the caller's default transaction.

`Transaction_Mgt.Start_transaction`
: Starts a local transaction.

`Transaction_Mgt.Abort_transaction`
: Aborts a transaction.

`Transaction_Mgt.Commit_transaction`
: Commits a transaction.

`Passive_Store_Mgt.Reserve`
: Reserves a passive version of an object on behalf of a transaction.

`Passive_Store_Mgt.Update`
: Updates an object's passive version.

You might think that the `Transfer` call is superfluous, since two successive calls to `Change_balance` would achieve the same outcome. This is only partly true, as the `Transfer` call, as described here, enforces atomicity of the transfer. This means, transactions ensure the call cannot charge one account and not credit the other.

First, both ADs, for the source and the destination account, are checked and amplified using the one-step `Access_Mgt.Import` call:

```

    source_untyped := Access_Mgt.Import(
        AD     => source_untyped,
        rights => change_rights,
        tdo    => account_TDO);
    dest_untyped := Access_Mgt.Import(
        AD     => dest_untyped,
        rights => change_rights,
        tdo    => account_TDO);
```

Next, the call makes sure that there is a default transaction. Note, that if the caller already started a transaction, no further transaction is needed.

The call reserves both objects. Time stamp conflicts are handled the same way as described in the previous section, with a program block with exception handler inside a loop. The following excerpt illustrates the two `Reserve` calls.

```
        Passive_Store_Mgt.Reserve(source_untyped);
        Passive_Store_Mgt.Reserve(dest_untyped);
```

Note that if the first `Reserve` succeeds but the second one fails, `Reserve` will be called again on both objects. At that point the `Reserve` call on the first object simply results in no operation.

After both objects have been reserved, the balances are checked. As the following excerpt shows, an insufficient balance in either account will will cause the `insufficient_balance` exception to be raised.

```
      if source_rep.balance - amount < zero
         or else
         dest_rep.balance + amount < zero
         then
        RAISE insufficient_balance;

      else
        source_rep.balance :=
            source_rep.balance - amount;
        dest_rep.balance   :=
            dest_rep.balance + amount;
        Passive_Store_Mgt.Update(source_untyped);
        Passive_Store_Mgt.Update(dest_untyped);
        if trans then
          Transaction_Mgt.Commit_transaction;
        end if;
        RETURN;

      end if;
```

The last step in a successful completion of the call, as shown in the example above, is to update both objects. The new balances do not become permanent until both objects have been successfully updated and the default transaction committed. Note, that even though the variables `source_rep_balance` and `dest_rep_balance` have been assigned the new balances, this has no effect on the passive versions of the objects unless they are updated from the active versions.

VII-6.2.6 Implementing the `Destroy_account` Call

The `Destroy_account` call destroys an account's passive version, and removes the master AD if it is stored with a pathname.

Calls Used:

`Access_Mgt.Import`
: Checks type rights and amplifies rep rights in one step.

`Transaction_Mgt.Get_default_transaction`
: Returns the caller's default transaction.

`Transaction_Mgt.Start_transaction`
: Starts a local transaction.

`Transaction_Mgt.Abort_transaction`
: Aborts a transaction.

`Transaction_Mgt.Commit_transaction`
: Commit a transaction.

`Directory_Mgt.Get_name`
: Returns the pathname of an object's master AD.

`Directory_Mgt.Delete`
: Deletes a directory entry.

`Destroy_account` uses the same techniques described in the previous sections to amplify rights on ADs and keep data in passive store consistent. The following example illustrates that after reserving the object's passive version, then if the balance in the account is zero, it calls `Passive_Store_Mgt.Destroy` to remove the object's passive version. If the object has no passive version, then the `Passive_Store_Mgt.no_master_AD` exception is raised.

```
Passive_Store_Mgt.Reserve(account_untyped);
if account_rep.balance /=
    Long_Integer_Defs.zero then
  RAISE balance_not_zero;
end if;
Passive_Store_Mgt.Destroy(account_untyped);
```

Finally the call attempts to remove the object's master AD. The following excerpt illustrates how:

```
loop
  declare
    path_text:  System_Defs.text(path_length);
  begin
    Directory_Mgt.Get_name(
        obj  => account_untyped,
        name => path_text);  -- out.
    if path_text.length >
       path_text.max_length then
      -- Text was lost.  Retry:
      path_length := path_text.length;
    else
      Directory_Mgt.Delete(path_text);
      EXIT;
    end if;
  exception
    when Directory_Mgt.no_name =>
      EXIT;
  end;
end loop;
```

If the master AD is (1) not stored in a directory, or (2) is stored in a standalone directory that does not have an associated name mapper, or (3) is stored in a standalone directory whose associated name mapper does not support `Get_name`, the call to `Directory_Mgt.Get_name` may fail and return with the `Directory_Mgt.no_name` exception.

Note that `pathlength` has an initial value of `60`. In the event that the pathname is longer than 60 characters, the loop body will be executed again, and this time around the `path_text` text record is declared with the actual length of the pathname.

In the last step the master AD will be deleted by calling `Directory_Mgt.Delete`. A master AD for the object may remain if other directory entries on the same volume set references the object. One of these alias AD will then become a new master AD.

VII-6.2.7 Initializing the Type Manager

In Section VII-6.1.1.1 we have discussed the need of the TDO to outlive any of its objects. For this reason the TDO has to be created and stored before the first call to this implementation of `Account_Mgt_Ex`. The TDO can be created either by the system administrator using the `configure` utility at node initialization time or by a separate procedure. In this chapter we shall discuss the second alternative. For more details on the first alternative, refer to the *BiiN™ Systems Administrator's Guide*.

Calls Used:

`Object_Mgt.Create_TDO`
: Establishes a new type by creating a new *type definition object* (TDO).

`Attribute_Mgt.Store_attribute_for_type`
: Stores an attribute with a TDO.

`Transaction_Mgt.Get_default_transaction`
: Returns the caller's default transaction.

`Transaction_Mgt.Start_transaction`
: Starts a local transaction.

`Transaction_Mgt.Abort_transaction`
: Aborts a transaction.

`Transaction_Mgt.Commit_transaction`
: Commit a transaction.

`Directory_Mgt.Store`
: Stores an AD with a pathname.

`Passive_Store_Mgt.Request_update`
: Requests an update of a passive version. No rep rights required.

The example described in this section is the `Stored_Account_TDO_Init_Ex` procedure. (The complete code of this procedure can be found in Appendix X-A.) This procedure has to be executed before `Account_Mgt_Ex` can be linked. Note also, that a TDO uniquely identifies its type. Calling the initialization procedure creates a new TDO that defines a new distinct type. You have to make sure that at any time there is only one passive version of the TDO on the system and that all instances of `Account_Mgt_Ex` refer to the same TDO, otherwise these instances will not be compatible.

The following excerpt from the `Stored_Account_TDO_Init_Ex` procedure shows how to declare the TDO and an instance of the *passive store attribute*.

```
account_TDO:  Object_Mgt.TDO_AD;
  -- TDO for accounts.

passive_store_impl:
    Passive_Store_Mgt.PSM_attributes_AD;
  -- Implementation of passive store attribute
  -- for accounts.
```

The next step is to create the TDO, to dynamically allocate an instance of the passive store attribute, to initialize the instance, and to store it with the type:

```
passive_store_impl := new
    Passive_Store_Mgt.PSM_attributes_object;

passive_store_impl.reset :=
    Refuse_reset_active_version_Ex.
        Refuse_reset_active_version'subprogram_value;

passive_store_impl.copy_permitted := false;

Attribute_Mgt.Store_attribute_for_type(
    tdo       => account_TDO,
    attr_ID   => Passive_Store_Mgt.PSM_attributes_ID,
    attr_impl => Untyped_from_PSM_attributes(
                     passive_store_impl));
type_name_impl := Account_Type_Name_Ex'package_value;

Attribute_Mgt.Store_attribute_for_type(
    tdo       => account_TDO,
    attr_ID   => Type_Name_Attribute_Ex.
                 Get_type_name_attr_ID,
    attr_impl => type_name_impl);
```

Note that the `passive_store_impl.reset` variable is initialized with a pointer to a subprogram that executes when `Passive_Store_Mgt.Request_reset_active_version` is called. The following excerpt from the `Refuse_reset_active_version_Ex` package in Appendix X-A shows this procedure:

```
procedure Refuse_reset_active_version(
    obj:  System.untyped_word)
is
  --
  -- Function:
  --   Handles requests to reset an account's active
  --   version by refusing such requests.
  --

begin

  RAISE System_Exceptions.operation_not_supported;

end Refuse_reset_active_version;
```

Note, that this procedure simply raises the `System_Exceptions.operation_not_supported` exception.

In addition, the `copy_permitted` boolean is set to false. This prevents a caller to duplicate accounts. The `Attribute_Mgt.Store_attribute_for_type` links the instance of the passive store attribute to the TDO. This operation does not, however, passivate the attribute instance. The next excerpt from the initialization procedure shows how the TDO and the attribute instance are explicitly stored:

```
if Transaction_Mgt.Get_default_transaction =
    null then
  Transaction_Mgt.Start_transaction;
  trans := true;
end if;

begin
  Directory_Mgt.Store(
      name   => account_text,
      object => Untyped_from_TDO(account_TDO),
      aut    => authority);
  Passive_Store_Mgt.Request_update(
      Untyped_from_TDO(account_TDO));
  Passive_Store_Mgt.Request_update(
      Untyped_from_PSM_attributes(
          passive_store_impl));
  Passive_Store_Mgt.Request_update(
      type_name_impl);

  if trans then
    Transaction_Mgt.Commit_transaction;
  end if;
exception
  when Directory_Mgt.entry_exists =>
    if trans then
      Transaction_Mgt.Abort_transaction;
    end if;

  when others =>
    if trans then
      Transaction_Mgt.Abort_transaction;
    end if;
    RAISE;

end;
```

Note again the use of transactions to ensure consistency of passive store.

VII-6.2.8 Protecting the Type Manager

Recall for a moment two premises of the type manager model:

- A type manager protects objects of its type.
- A type manager provides black box type functionality.

In order for your type manager to accomplish these requirements you have to properly protect it from other programs. There are two aspects to protecting the type manager, namely

- protecting the type manager inside a running program,
- protecting the type manager's private ADs,

Calls Used:

`Authority_List_Mgt.Create_authority`
Creates an authority list.

`Identification_Mgt.Get_user_ID`
Returns caller's user ID.

Protecting the type manager inside a running program is equivalent to protecting its address space. The BiiN™ Systems Linker provides special support for linking modules so that each one executes in its own protected address space, called *domain*. Besides creating an executable program, you can also create an *image module* with the linker. Image modules are pre-linked pieces of software that are not linked to a user's program until runtime and that can be shared by several users. An image module always executes in its own domain. For more details on domains and image modules, in particular on how to build domains and image modules with the linker, refer to the *BiiN™ Systems Linker Guide*.

Depending on how your type manager is to be used, you can choose to either link it in the standard way to an interactive interface, or to link it into an image module, thus making it available to be called by user programs. If the type manager consists of small routines that are not going to be called very often, the savings of shared code will not outweigh the overhead of creating an image module. For large programs used frequently, however, using image modules could result in substantial savings.

The second aspect of protecting the type manager is to protect its private ADs. It is necessary for the protection mechanism here that the linking not be left to the user for the following reason: As mentioned above, you need to create and store the TDO before invoking the type manager for the first time. The TDO is created by an initialization routine that stores it with a pathname. This directory entry is protected by an authority list. The following excerpt from `Stored_Account_TDO_Init_Ex` is an example where the authority list includes only the caller.

```
owner_only:  User_Mgt.protection_set(1);
  -- Protection set that includes only one ID, namely
  -- the type manager's owner.

authority:  Authority_List_Mgt.authority_list_AD;
  -- Authority list that contains only one ID, namely
  -- the type manager's owner.
. . .
 owner_only.length := 1;
 owner_only.entries(1).rights := User_Mgt.access_rights'(
     true, true, true);
 owner_only.entries(1).id := Identification_Mgt.Get_user_id;

 authority := Authority_List_Mgt.Create_authority(owner_only);
. . .
  Directory_Mgt.Store(
      name   => account_text,
      object => Untyped_from_TDO(account_TDO),
      aut    => authority);
```

The TDO is retrieved at link-time using the Ada pragma `bind`. At that time rights are evaluated against the ID list of the calling process. The following excerpt from the implementation shows this:

```
account_TDO:  constant Object_Mgt.TDO_AD := null;
  -- This is a constant AD but not really null; its
  -- filled in with an AD retrieved by the linker.
  pragma bind(account_TDO,
              "account");
    -- Bind to TDO for accounts.
```

With the TDO thus protected, only people who are included in the TDOs authority list can link the program since noone else has access to the TDO. In the above example this is only you. (You could also create a separate ID just to protect the type manager.)

After the program is linked, it can execute with any ID.

VII-6.3 Summary

In this chapter you have learned the techniques necessary to build a type manager for stored objects. In particular, you have learned that

- before the first object can be stored, a TDO has to be created and stored together with a list of attributes.
- storing an object requires two steps, namely storing the AD and updating the object's representation.
- the use of transactions keeps passive store consistent even in the event of a system failure.
- transactions can be used to synchronize access to passive objects.
- removing an object that has been passivated requires three steps, namely, deallocating the active version, destroying the passive representation, and deleting the master AD.
- special features of the linker and pragma `bind` can be used to protect the type manager.

NOTE

Please keep in mind that the example described in this chapter permits processes in different jobs to concurrently use the objects of one type. There is no provision in the example for processes within one job to concurrently access one object. For details on how to achieve that, see Chapter VIII-1.

UNDERSTANDING SYSTEM CONFIGURATION 7

Contents

A *configuration* is an arrangement of objects representing the hardware and software resources of a particular BiiN™ node. System administrators routinely manage node configuration using the `configure` utility as described in the *BiiN™ Systems Administrator's Guide.* Two classes of programmers also need to understand system configuration:

- Programmers adding hardware devices to BiiN™ systems
- Programmers adding software services with unique initialization requirements.

A BiiN™ system provides a variety of predefined system configurations describing systems covering the most common customer characteristics of hardware configuration: number of users, interactive or batch workload, or computational or I/O emphasis. Any of these predefined configurations may be used for generating a tested and balanced BiiN™ Operating System configuration, or may be modified to accommodate site-specific requirements.

Packages Used:

`Configuration` Provides operations for creating and modifying a system configuration.

Configuring a system includes creating configurable objects to represent hardware and software system components, then attaching and starting the objects to build a running system.

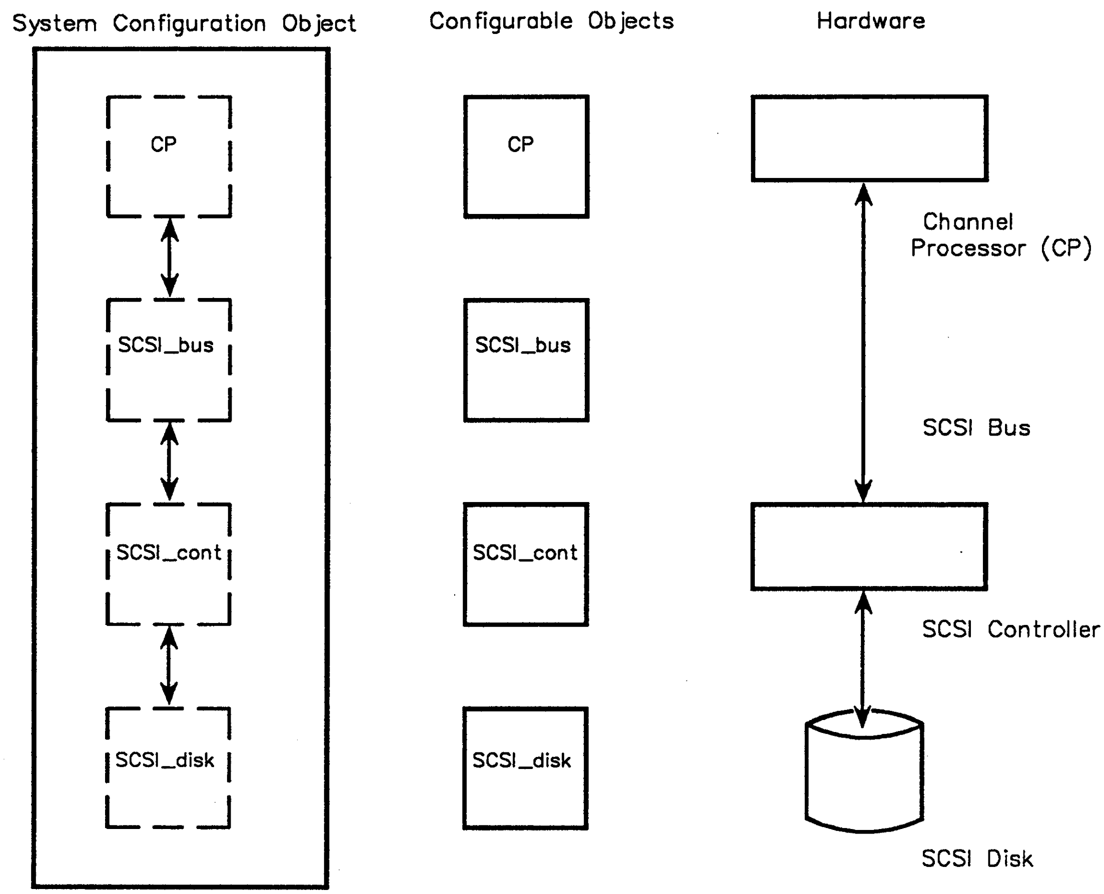

Figure VII-7-1. System Configuration

VII-7.1 Creating a Node's Configuration

A node's configuration is created when the node is booted (see Figure VII-7-2). Booting a node begins with all hardware connections made, power on, and needed boot images but no software active in the system. Booting ends with a functioning, active system ready to respond to commands. The boot process must search for and initialize hardware and software modules and create the complex network of objects on which a running node depends.

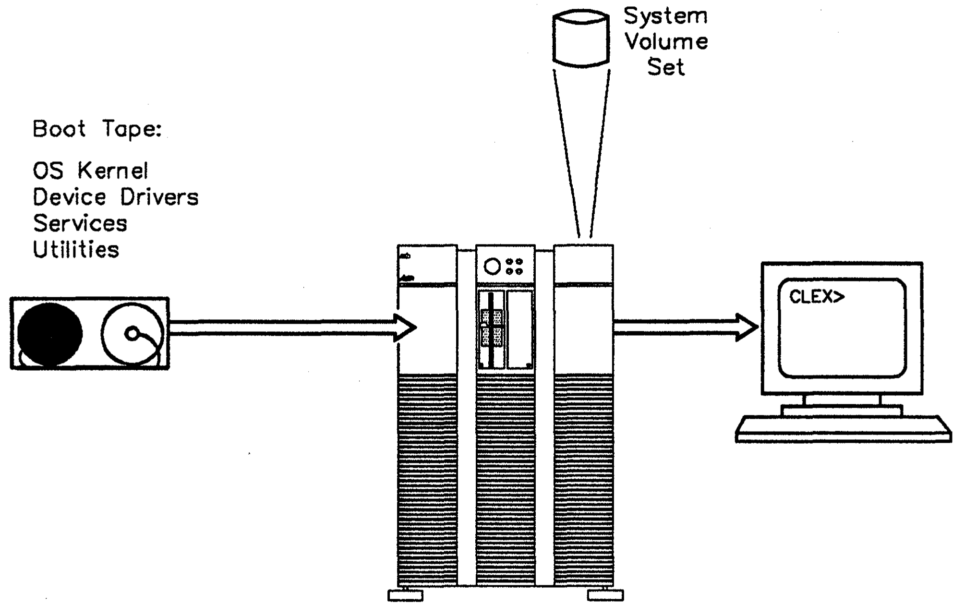

Figure VII-7-2. Booting a Node

Certain information must be available when a node is configured:

- What objects are part of the configuration. For example, there may be objects that represent physical I/O devices, device controllers, logical devices such as volume sets, and software units such as the OS kernel.
- One-time operations to be performed. For example, a hard disk may need to be formatted.
- The sequence in which operations should be performed. For example, a volume set cannot be created on a hard disk until after the disk controller is started and the disk is formatted.

VII-7.2 Defining a Node's Configuration

A node's configuration is defined by a *System Configuration Object (SCO)*. An SCO provides information needed to create the configuration: the objects involved, the operations involved, and the required sequence of operations.

An SCO is a list of operations to perform, along with parameters for each operation. Only those operations defined by the `Configuration.Ops` attribute package are allowed in an SCO. If an object type needs to actively participate in the configuration process, that type must support the configuration attribute. Such objects are *configurable*.

VII-7.3 Configuration Attribute Calls

The configuration attribute provides calls for:

- *Attaching* objects to configurable objects
- *Starting* configurable objects.

These calls are normally used within an SCO. Other configuration attribute calls, for *detaching* objects from configurable objects and *stopping* configurable objects, are normally not used within an SCO.

VII-7.4 Creating Configurable Objects

System configuration is the specification of environmental hardware and software operating parameters of the components to be supported by a BiiN™ Operating System kernel image. *System components* include hardware modules (disk, controller, bus, etc.) and software modules (loadable, non-resident subsystems and optional support services).

A *configurable object* (CO) is a representation of a hardware or software module that must be configured at node initialization, or can be dynamically added to a running node. A *configuration attribute* supports the configuration of objects other than software services, particularly hardware components. A *service configuration attribute* supports the configuration of software services that have configuration and initialization dependencies in common. (An object is configurable only if its TDO contains the configuration or service configuration attribute.)

A configurable object must be created for each system component to be included in a system configuration. After it is created, it is not yet functional, but may be attached to other configurable objects. Attachment binds the configurable objects so they can be started and placed in a usable state.

When the configurable objects are no longer required to provide their function, they can be stopped. When they are no longer needed in the configuration, they are detached from other configurable objects to which they may have been attached.

Figure VII-7-3 illustrates the process of creating a configurable object.

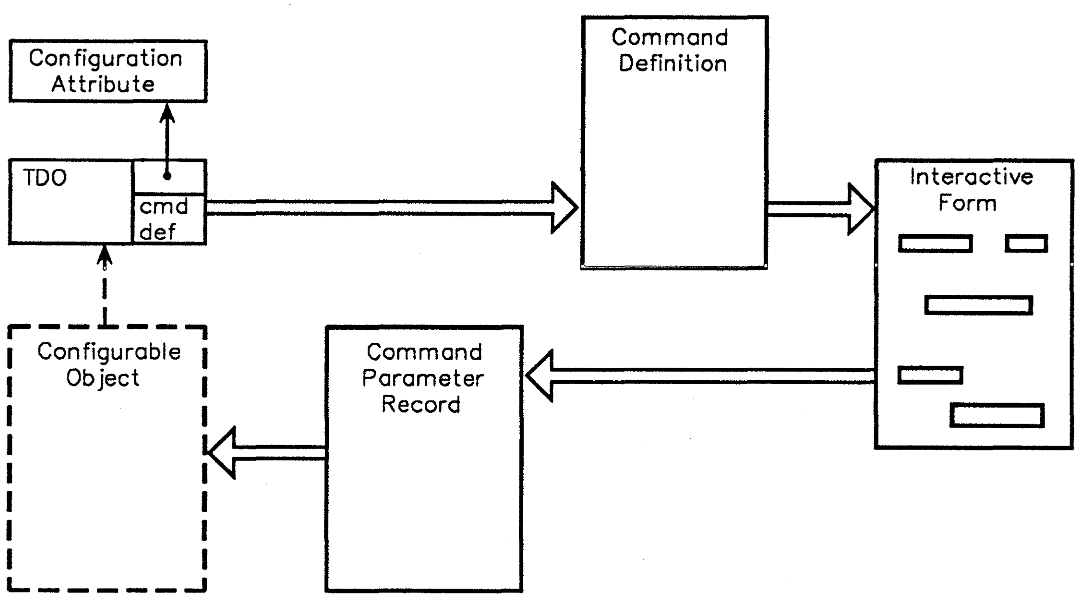

Figure VII-7-3. Creating Configurable Objects

An object to be made configurable must have a TDO which contains a configuration attribute. The TDO contains a command definition that defines the type of information required by a configurable object of the TDO's type. This command definition is displayed in an interactive form through which a user enters parameter data. The data collected by the interactive form is extracted from the command definition format and is used to create a configurable object.

VII-7.5 Attaching Objects to Configurable Objects

`Attach` and `Detach` operations bind and unbind configurable objects. These configurable objects are considered head or tail objects depending on their relationship in the binding.

A *head object* is the initiating member of a pair of configurable objects associated with each other. A head object is characterized by its ability to function normally without being attached to another configurable object.

A *tail object* is the dependent member of a pair of objects associated with each other. A tail object is characterized by the requirement to be bound to a configurable object before it can become functional. Rights that may be needed on tail objects should be specified by the type manager supporting the `Attach` and `Detach` configuration calls on the tail objects. Tail objects don't have to be configurable when the attachment is unidirectional (tail object attached to head object but head object not attached to the tail object).

An attachment normally indicates that the tail object depends on the head object to function. For example, a volume set must be attached to a disk in order to function. A type manager's implementation of `Attach` normally checks the validity of the attachment by checking the type, rights, and state of the tail object and the rights and state of the head object.

An implementation of `Attach` can be bidirectional, making the attachment in the reverse direction as well. A bidirectional implementation is used when configurable objects are mutually dependent. For example, a CP (channel processor) and a SCSI (Small Computer System Interface) bus must communicate with each other in both directions and therefore require a bidirectional implementation of `Attach`.

VII-7.6 Configuring Software Services

A configurable object is an object whose TDO contains an instance of a configuration attribute. Kernel, loadable, and application services require an attribute that can deal with the interdependencies inherent between them. For example, the object service uses the distribution service which in turn uses the clearinghouse service. An attribute is provided by configuration that, for example, enables the distribution service to ensure that the object service is started only after the Clearinghouse is started.

The mechanism used to support this binding of services is the *service configuration attribute.* This attribute allows a service to link itself with all the necessary and optional services that it uses. This attribute is extensible in that it allows a service to support the initialization of services that use it, and allows a service's initialization to itself to depend on other services. This attribute registers a distribution service-dependent initialization procedure. These procedures are called by the BiiN™ Operating System after the system SCO has been processed when a node is present in a distributed system.

VII-7.7 Starting Configurable Objects

All configurable objects provide `Start` and `Stop` implementations (which can be null). `Start` places a configurable object into a usable state by performing local initialization. `Start` is called by OS initialization as specified in a System Configuration Object (SCO). `Start` can also be called to start a component in a running system. Starting a configurable object should not start any attached tail objects. However, `Start` may require that tail objects be already started.

When the object to be started is a configurable object (CO) or a software service (SS) that neither is dependent on another software service nor is depended on by another software service, `Start` places it into a usable state by performing local initialization.

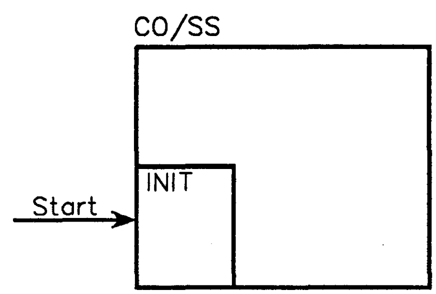

Figure VII-7-4. Simple Attach

When the object to be started is a software service that is dependent on another software service, `Start` performs local node initialization and attaches the first software service to the service on which it is dependent.

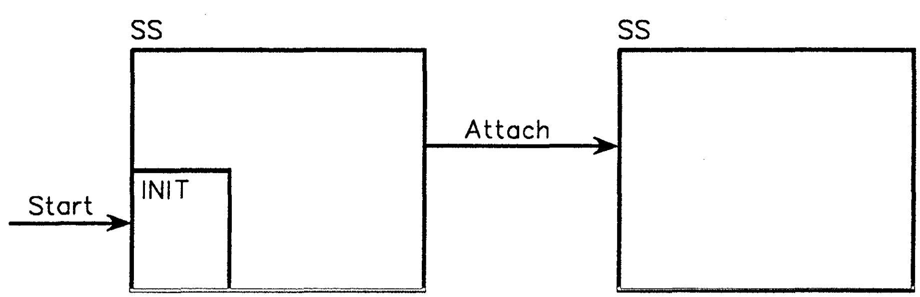

Figure VII-7-5. Attaching to a Dependent Software Service

When the object to be started is a software service that another service depends on, `Start` performs back attaches, that is, attaches the dependent service to the service that it depends on.

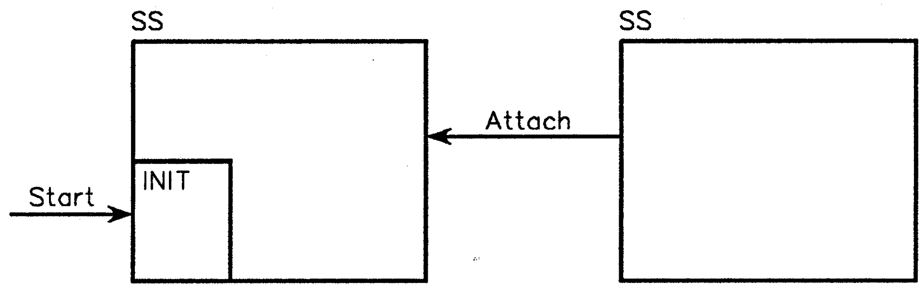

Figure VII-7-6. Back Attachment of a Dependent Software Service

When the object to be started is a software service (A) that is both dependent on another software service (B) and another service (C) depends on it, `Start` first attaches A to B on which it is dependent, and then performs back attaches from A to C.

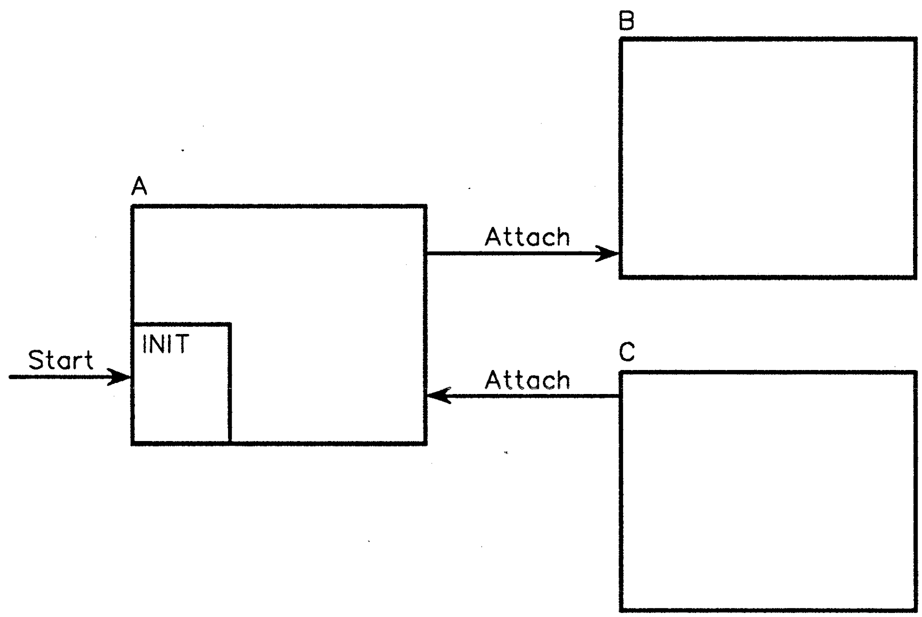

Figure VII-7-7. Compound Attachment

The order of attaches caused by starting a software service is implementation-dependent.

VII-7.8 System SCOs and User SCOs

A *System Configuration Object* (SCO) is composed of a sequence of commands that attach COs together and start COs. The system administrator specifies a system SCO and a user SCO to use during OS initialization. A *system SCO* references hardware and software components of the configuration that are required to complete the node's initialization of the BiiN™ Operating System. A *user SCO* references components of the configuration that are not required to complete initialization of the OS, such as starting login services, database systems, specific application programs, and other activities that depend on disk write access or distributed system services.

Figure VII-7-8 illustrates system and user SCOs:

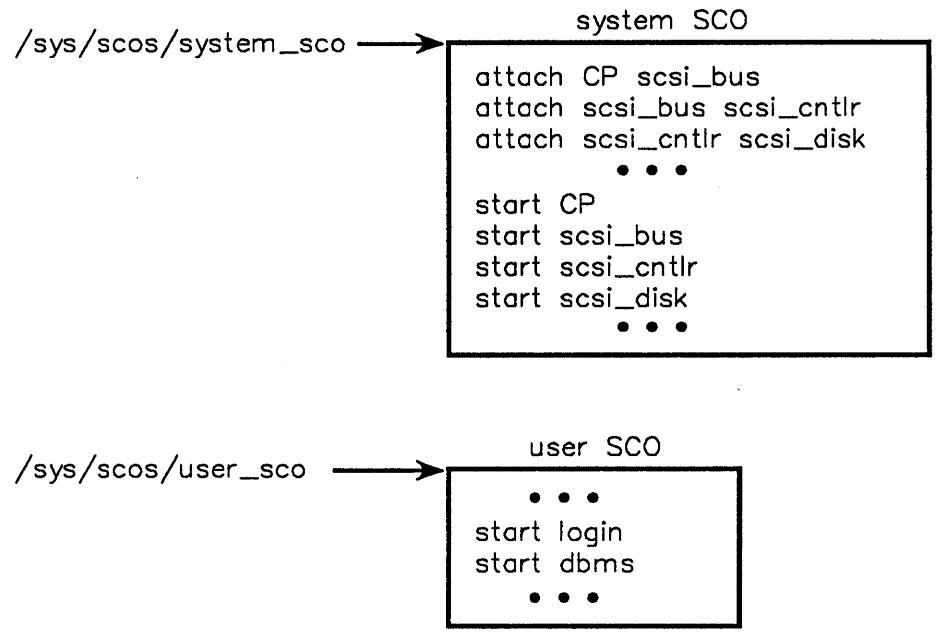

Figure VII-7-8. System Configuration Objects

The order of initialization of configurable objects is defined by the sequence of `Start` calls in the SCOs. The sequence for other configurable objects started after system initialization is determined by their type managers. For example, a set of configurable objects that is part of a CP (Channel Processor) subsystem can be started by starting the configurable object that represents the CP. Conversely, various network services require a separate start for each service specified in the configuration.

All system and user SCOs on a node are contained on the system volume set in the directory `/sys/scos`.

VII-7.9 The `configure` Utility

Additional system configuration can be performed dynamically when the system is up and running, or at the next boot by updating or creating new SCOs.

The `configure` utility provides runtime commands to dynamically attach, detach, start and stop COs, and to create COs and SCOs for use at a future system initialization. See the *BiiN™ Systems Administrator's Guide* for information about the `configure` utility.

VII-7.10 Summary

- Hardware components and system software modules are defined to represent a working system.
- A running system can be modified with the `configure` utility to build a site-specific system.

- System configuration is the specification of environmental hardware and software operating parameters of the components to be supported by a BiiN™ Operating System kernel image.
- System configuration is the process which brings a nonfunctional system to the point that it can execute a common application.
- *System components* include hardware modules (disk, controller, bus, etc.), and software modules (loadable, nonresident subsystems, and optional support services).
- A *configurable object* (CO) is a representation of a hardware or software module that must be configured at node initialization or can be dynamically added to a running node.
- A *System Configuration Object* (SCO) is composed of a sequence of commands that attach COs together and starts COs.
- When a system is up and running, additional system configuration can be performed dynamically, or at the next boot by using the `configure` utility.
- A *service configuration attribute* enables a service to link itself with all the necessary and optional services that it uses.

Part VIII

Distribution Services

This part of the *BiiN™/OS Guide* describes OS support for distributed services.

The chapters in this part are:

Understanding Distribution
Explains basic concepts of distribution and distributed services.

Building a Distributed Type Manager
Explains how to build a local single-activation distributed type manager, using remote procedure calls.

Distribution Services contains the following services and packages:

clearinghouse service:

```
CH_Admin
CH_Client
CH_Support
Node_ID_Mapping
```

RPC service:

```
RPC_Admin
RPC_Call_Support
RPC_Mgt
```

transport service:

```
Comm_Defs
Datagram_AM
DG_Filter_Mgt
Distributed_Service_Admin
Distributed_Service_Mgt
ISO_Adr_Defs
ISO_Config_Defs
ISO_TM_Admin
TM_Comm_Defs
VC_Filter_Mgt
Virtual_Circuit_AM
```

Part VIII Overview

UNDERSTANDING DISTRIBUTION 1

Contents

VIII-1.1 Introduction

The BiiN™ Operating System supports distributed computing. A distributed system, capable of distributed computing, spans a number of BiiN™ nodes connected by a communication network. The network may contain several subnetworks. In this context a subnetwork is a homogeneous network such as ethernet or HDLC. It is important to note that the network connecting a distributed system need not be homogeneous. Two distributed system may also share a homogeneous subnetwork, such as a LAN (local area network), for example. Distribution is a high level concept independent of the communication media and associated communication protocols. Although distribution is independent of the communication media, it is optimized for high speed LAN applications.

A distributed system may appear as a ''single machine'' to the casual user. On the other hand a user can use his/her knowledge of the structure of the system, and work with individual or defined collection of components (nodes, I/O devices, and so on).

Figure VIII-1-1 shows an example of a network of BiiN™ nodes.

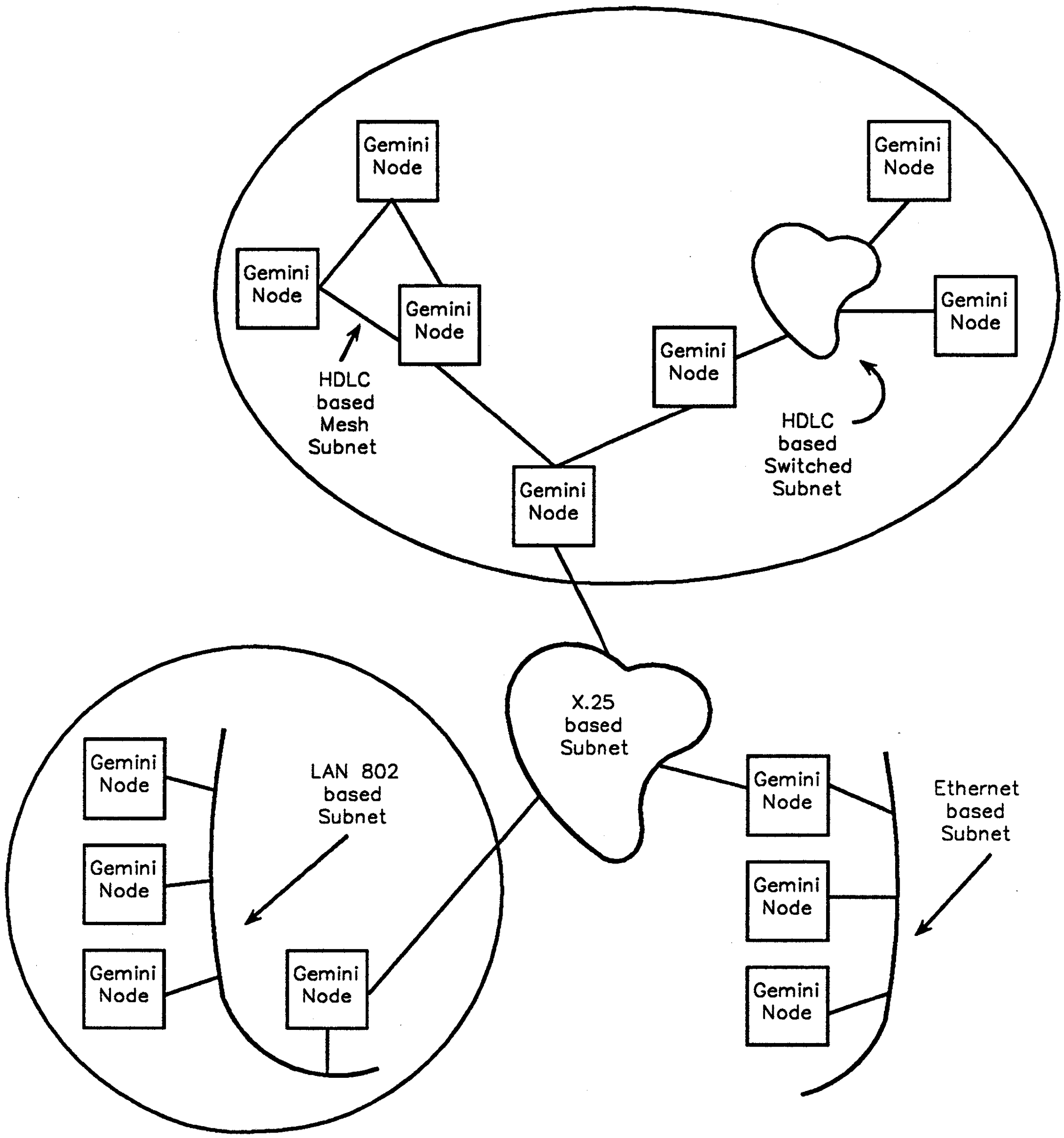

Figure VIII-1-1. A Network of BiiN™ Nodes

This particular network contains two bus-based LANs connected via a public packet switched network. Two additional subnetworks are shown, one based on a set of dedicated point to point communication lines and the second based on a circuit switched network. Circles indicate the boundaries of distributed systems.

Distributed computing lies in between *multiprocessing* and *networking*. Table VIII-1-1 lists important points in which the three concepts differ.

Table VIII-1-1. Distribution vs. Multiprocessing vs. Networking

Multiprocessing	Distributed Computing	Networking
Close Cooperation	Cooperation	Mutual Suspicion
Complete Trust	Tempered Trust Access/Resource Controls	No Trust
Single Administrator	Cooperating Administrators	Independent Administrators
Completely Shared Resources	Controlled Sharing of Resource	No Shared Resources
"Single Machine"	Homogeneous	Heterogeneous

On one hand distribution extends the concepts of multiprocessing beyond the limits of one shared memory, and on the other hand distribution takes the ideas of networking one step further.

This chapter explains the concepts of distribution. It does not explain specific techniques or point out the details of implementing a distributed service. This information is contained in chapter VIII-2.

The next section gives examples of what a distributed system can do and what it cannot do. The following sections discuss the most important aspects of distribution in more detail, in particular the following topics:

- Communications
- Naming
- Review of the computational model
- Single activation distributed services
- Protection in a distributed system
- Transparently distributed services.

Communications and naming are the two building blocks of the distributed architecture. For this reason special attention will be given to these two areas.

VIII-1.2 What a Distributed System Can Do

Distributed computing makes it possible to build computer systems of any size from a single node up to a conglomerate of as many nodes as you choose. (There is no limit to the size of a distributed system.) Even though only a conglomerate of individual machines, the system acts in many ways as if it were one single machine, provided, of course, that the communication media is fast enough.

In most cases the user need not be aware of the physical organization of the distributed system; although nodes are individual machines that can operate by themselves, they appear to the casual user to be one unit. For instance, disks are mounted on individual nodes, but they appear to be mounted on all nodes at once. A user can also choose to run a job on a selected node or to store an object on a particular disk drive of his/her choice.

Jobs are the computational unit in a distributed system. Jobs run on single nodes but they communicate with other jobs, on the same node or on other nodes in the system. The interface for job communication on different nodes and the same node is identical, but there is an efficient implementation of intra-node communications.

By the means of interjob communication, independent jobs may exchange messages or related jobs may be coupled together. A *service*, such as the filing service, may contain jobs that run concurrently on all nodes of the system. The service is thus available on all nodes. All jobs belonging to the service communicate constantly and create a homogeneous environment of file access and usage across the entire system: Any file on the system is uniquely identified and stored in one place; this avoids a considerable amount of duplication. Files are available from any node: Requests to access a file are forwarded to the file's home node and executed there.

The filing service is a *universal service.* Universal services are decentralized; filing requests are serviced on the node where the requested file is stored. Since files can be stored at any node, filing services requests on all nodes of the system. (Diskless nodes are currently not supported.)

Services can also be *regional.* A regional service is centralized; requests can be issued on many nodes but only a few nodes (or even a single node) service requests. Universal services are "symmetric"; on all nodes there is an *agent* that accepts and distributes requests and a *server* that receives requests from an agent and executes them. A regional service is "asymmetric"; there are many agents and only a few servers.

Compare a universal service to the postal service: Every town has its own post office that receives mail from other towns, distributes it to the addressees, and collects and processes outgoing mail. A regional services resembles more an insurance company. Insurance agents sell policies for a company that underwrites the policies. The agent interacts with the clients on the one side and with the insurance company on the other. The insurance agent does not underwrite policies himself.

As an example of a regional service imagine an airline reservation system. All booking information is kept in a few locations. Agents in branch offices make reservations on their local nodes; the requests are transparently forwarded to one of the nodes where booking information is kept.

Distributed systems provide parallel processing. A *session* may span several nodes and contain jobs on all those nodes. If a task can be partitioned, processes in these jobs can work on parts of the task asynchronously.

Currently, load balancing is not implemented. The architecture does not discourage this functionality, however. An application implemented as a distributed service can decide based on the load in the system, how it routes requests to its servers. An example is a distributed batch utility that submits batch jobs to the node with the lowest load in the system.

The following two sections discuss the most important elements in a distributed system, namely how entities are named, and how nodes in the system communicate.

VIII-1.3 Naming

One of the two building blocks of a distributed architecture is a location-independent naming mechanism. Here is an example of the merit of location-indepent naming: A volume set is identified on the machine level by a unique *volume set ID.* The volume set ID reflects where the volume set is currently mounted in the system. The symbolic name of the volume set on the other hand has nothing to do with the location of the volume set. More importantly, the symbolic name does not change when the volume set is moved to another node. You can refer to the volume set without having to know where it is currently located.

Naming extends to stored objects, users, nodes, and volume sets. The map from machine level identifiers to symbolic names is maintained the *clearinghouse*.

The clearinghouse centralizes network information in a few locations. Thus network information can be updated quickly and easily. Volume sets can be moved from one node to another, a node may be added, or a node may be disconnected: Those changes have to be recorded in only a few places, namely where copies of the clearinghouse are kept.

VIII-1.3.1 The Clearinghouse

The clearinghouse is decentralized and replicated. Instead of one global clearinghouse server there are many local servers each storing a copy of a portion of the global information. Some information in the clearinghouse is cached locally by other services. This allows to bypass the clearinghouse for efficiency and when access to a clearinghouse server is not possible due to a communication failure.

User ids, for example, are available at all nodes. This is necessary in order to allow users to log on to a local node even if that node is disconnected from the rest of the system. The same applies to locally mounted volume sets.

The organization of the clearinghouse is hierarchical. Names of clearinghouse entries consist of four parts representing the four level hierarchy. The names of the four parts are *organization, domain, environment,* and *local*. Clearinghouse names are specified with single, double and triple slashes between the level names. A full clearinghouse name is always of the following form:

```
///org/dom/env/local
```

Organization and domain together reference a `naming domain`. A large distributed system is typically split up into multiple naming domains. Thus name evaluation does not become hopelessly slow when the system becomes very large. Every node in the system belongs to exactly one naming domain. The clearinghouse is partitioned on the naming domain level. This means that one clearinghouse server stores all entries of the form

```
///organization/domain/anything/anything
```

A name starting with two slashes reference an entry in the callers organization:

```
//dom/env/local
```

A clearinghouse name starting with one single slash refers to the local naming domain:

```
/env/local
```

Figure VIII-1-2 illustrates the hierarchical structure of the clearinghouse.

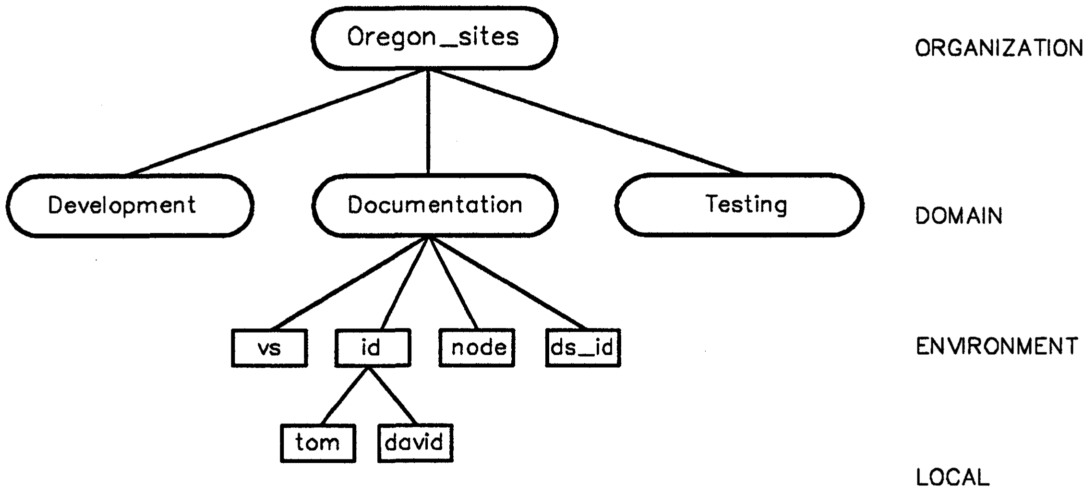

Figure VIII-1-2. The Hierarchical Structure of the Clearinghouse

The information in figure VIII-1-2 is shown together in one place. In a real system it is partitioned, replicated, and stored in different locations. The figure is very much simplified and shows entries for only one naming domain. This is done for convenience and ease of understanding.

There is one special naming domain per distributed system, called the *figurehead* naming domain. This domain covers the entire system. More specifically, it references all other entries in the clearing house. In fact, the figurehead naming domain defines the distributed system. It is used whenever the naming domain of an object is not known. This can happen when a passive object is activated: `Passive_Store_Mgt` has a *unique identifier* (UID) for the object which contains the ID of the volume set where the object is stored. With the help of the figurehead naming domain, `Passive_Store_Mgt` maps the volume set ID to the network address of the node where the volume set is mounted.

The clearinghouse is maintained by the clients, BiiN™ Operating System services or applications that use the clearinghouse. Clients maintain clearinghouse *environments*. In an environment the clients store names and properties associated with those names. The naming service, for example, maintains the `vs` environment. It uses this environment to map volume sets to node addresses, indicating where the volume set is mounted. Another example is the protection service. It maintains the `id` environment that maps user IDs to user profiles (and thus to symbolic user names). This information is used by the logon utility. The distributed OS services use a total of four environments in the clearinghouse, namely `vs`, `id`, `node`, and `ds_id`. From the point of view of the clearinghouse there is no difference between those environments and other environments. The clearinghouse simply provides the mechanisms for binding symbolic names to properties in one networkwide location. It is entirely up to the client to attribute meaning to the clearinghouse entries.

Most applications will use the clearinghouse indirectly through the OS services. However, if the need arises, an application may use the clearinghouse directly, either through the above mentioned environments or even by setting up its own environment.

A request to the clearinghouse to bind a name to a set of properties may originate anywhere in a distributed system. The request will be directed to a clearinghouse *agent*. The agent knows the address of at least one clearinghouse *server*. The server will either handle the request directly or, if it does not store the required information, forward the request further to a server that stores the information. This entire process happens invisibly to the client.

In summary the clearinghouse provides the basic tools needed for a high level naming mechanism. But the function of the clearinghouse goes beyond this task. Any type of information may be bound to a name; an internetwork address, in the case of a node, or a telephone number, in the case of a user. Services can use the clearinghouse to whatever purpose they require. The merit of the clearinghouse is that it centralizes all this information and makes it available to everyone. One of the most important uses of the clearinghouse is to provide location independent naming.

VIII-1.4 Communications

If distribution is compared to a brick wall, then naming corresponds to the bricks and communications to the mortar; either one without the other would be useless. And just as mortar and bricks become invisible once plaster has been applied, so should the details of naming and communications be invisible in a distributed system. However, nobody can build a wall without mortar, and nobody can build a distributed system without communication between nodes. In order to understand distribution, we have to have some understanding of how nodes communicate.

One of the guiding principles in the BiiN™ architecture is that logical structures hide physical structures. This principle also pertains to communications: The system supports a variety of different communication protocols, such as Ethernet, IEEE 802.3, HDLC and X.25. *Transport services* hide the details of these various subnetworks. Through the interfaces provided by transport services a distributed service can use two different high level communication protocols, a connection oriented and a connectionless protocol. We refer to the connection-oriented protocol as a *virtual circuit* and to the connection-less protocol as a *datagram.*

Datagrams are short one-way messages sent from one job to another. They are similar to letters sent through the mail: There is no guarantee that a datagram sent will be received by the addressee or that a number of messages sent will be received in the order that they were sent. Transport services only guarantee that if a message is received, it will be intact. On the positive side datagrams are inexpensive (just as letters), fast, and require little overhead.

Virtual circuits provide a full duplex connection between the connected parties. A virtual circuit is a bidirectional ordered flow of bytes similar to a telephone connection. Receipt of a message is acknowledged and messages sent in a certain order arrive at the addressee in that same order. Setting up, maintaining, and tearing down a virtual circuit presents considerable overhead.

There is a third way for processes to communicate. This method is called a *remote procedure call.* Remote procedure calls are built on top of datagrams and share some of the advantages of datagrams. They provide the following additional services:

A simple call interface
: Making an RPC involves no more than making an ordinary procedure call.

Authentication and security
: Messages are authenticated to insure that they are intended for that server and that they have not been modified in transit.

Converting ADs	ADs are converted to their passive form.
Locating	Given an AD to the server, RPC locates the server.

RPCs are message/reply pairs. They force the caller to wait until the call has completed. A series of RPCs made by one process is strictly ordered, since the calling process cannot make another RPC before the previous one has completed. RPCs are used within distributed services to communicate between instances of the service. (RPCs made by different processes in a certain order do not necessarily retain that order.)

It is important to note the conceptual difference between RPCs on one side and datagrams and virtual circuits on the other. RPCs use datagrams as means of communication, they provide additional services as mentioned above, and they are not as flexible as datagrams. RPCs are taylored specifically to the needs of distributed services. Datagrams and virtual circuits are basic means of communication and not taylored to any specific application. They provide no locating services, no authentication, and their interface is more complicated than RPCs. In exchange they can be used for any type of communication between jobs, not just between instances of a distributed service.

Whether an application uses RPCs, datagrams or virtual circuits depends on its particular needs. An application set up as a distributed service will find RPCs the easiest to use. For other uses datagrams or virtual circuits provide the necessary flexibility. In particular datagrams are good for sending brief messages, and virtual circuits for reliably transmitting large amounts of data.

Figure VIII-1-3 gives a simplified picture of the differences between datagrams, virtual circuits, and RPCs.

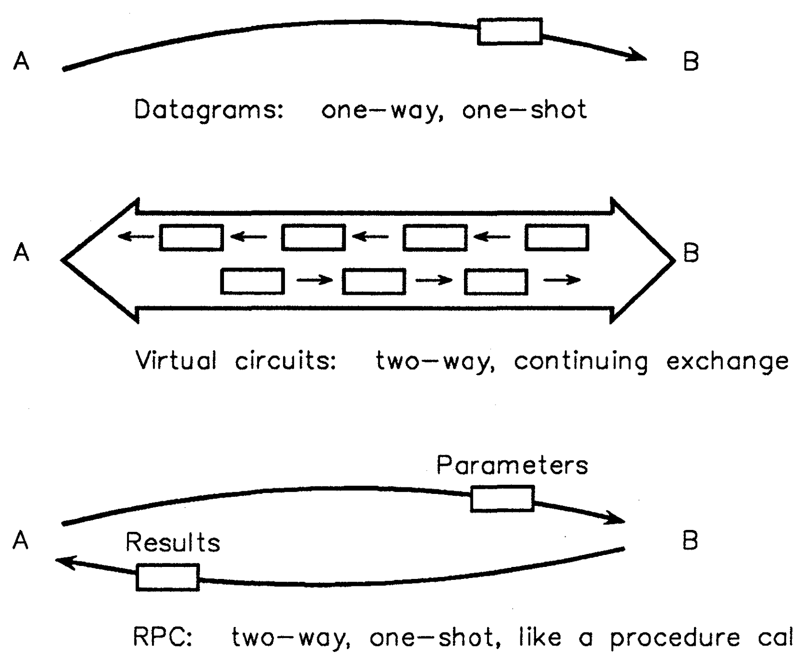

Figure VIII-1-3. Three Different Communication Methods

Both datagrams and virtual circuits link two jobs. To be more precise, datagrams are sent from one *transport service access point* (TSAP) to another. A TSAP represents a binding between the user of a transport service and the transport service itself. A TSAP object represents a TSAP. In the case of datagrams the TSAP object also serves as a repository for information relating to the TSAP that it represents. This includes buffers and state information. TSAPs are specific to either datagrams or virtual circuits.

In the case of a virtual circuit there is an additional, dynamic level of association between communicating processes, the connection. A transport connection point (TCP) represents an endpoint of the connection. In this case the TSAP represents only the static binding between user and transport service and is used to create and destroy TCPs which represent the dynamic binding. Multiple TCPs can be associated with one TSAP (but only one TSAP with any TCP).

TSAPs are bound to a *TSAP address*. A TSAP address uniquely identifies a TSAP over the entire network. A user who wants to send data through his TSAP to another TSAP must know the TSAP address of the destination TSAP. The remote user can receive the data on his TSAP along with the sender's TSAP address.

TSAP addresses are composed of two parts, a network part which uniquely identifies an instance of the transport services, typically associated with one node, and a *transport service end point*. The network part is known as an *NSAP*. An NSAP is the point at which an instance of the transport services is bound to the network level services. Inside the realm of an NSAP an end point uniquely identifies a TSAP.

It is convenient for some system-wide services to reserve certain fixed values of end points. Those end points are called *well known endpoints*. Other endpoints are dynamically allocated by the transport services.

Summarizing, the BiiN™ architecture provides high level interfaces for communications between nodes in a distributed system. Depending on the needs of an application communication services can be used at different levels. However, at all those levels an application does not have to be concerned with the details of the communication protocol.

VIII-1.5 Review of the Computational Model

In the previous two sections we have outlined naming and communications in a distributed system. Those are the building blocks for a distributed architecture. In this section we shall review the BiiN™ computational model briefly and put it in perspective in a distributed system.

VIII-1.5.1 Processes, Jobs and Sessions

Processes represent linear threads of computation. Multiple processes may be part of one job. Jobs are the unit of program execution in the BiiN™ system. Jobs, and therefore processes, are confined to a single node. A session may contain many jobs on different nodes. The jobs in the session can communicate with each other or with jobs outside their session. In many ways a job acts like a virtual computer.

VIII-1.5.2 Active and Passive ADs

Active access descriptors (active ADs) are represented by 33bit words where the 33rd bit, the tag bit, is set. Active ADs are valid inside a node's active memory only. Before an AD can cross node boundaries in a distributed system, it has to be converted to its passive version. A passive AD is a much larger entity than an active AD (about 40bytes). A passive AD is a unique reference on all BiiN™ systems at all times. In order for an object to have a passive AD an AD to the object has to have been stored previously.

VIII-1.5.3 Single and Multiple Activation Model

The system supports two different models of activating passive objects (copying passive objects into active memory). In the *multiple activation model* any job activating an object receives an independent active copy of the object. A job can work on its copy and update the passive version from the active version. The multiple activation model is easy to use except for one problem; passive store refuses updates from outdated versions. A job whose update has been refused can handle this situation by requesting a fresh active version, redoing its changes, and attempting another update.

The single activation model avoids the updating problem by allowing only one copy of an object in active memory. One job, the *home job*, receives the active version and all other jobs receive stand-ins, called homomorphs, when activating an object. Those jobs who have homomorphs communicate with the *home job* in order to effect changes on the object. The single activation model is useful for large objects that are used by many jobs simultaneously.

There is an important difference between how global and local objects are treated in both the single and the multiple activation models. Independent of whether in the single or multiple activation model there is always a maximum of one active version per of node of a global object. All jobs accessing the global object share this one active version. In the single activation model there is one active version of an object per distributed system, in the multiple activation model there is one active version per node of a global object, and one active version per job of a local object. Independent of the activation model processes within one job always share an active version

Figure VIII-1-4 illustrates the difference between single and multiple activation model. Note that what is shown as active memory in the figure may span several nodes.

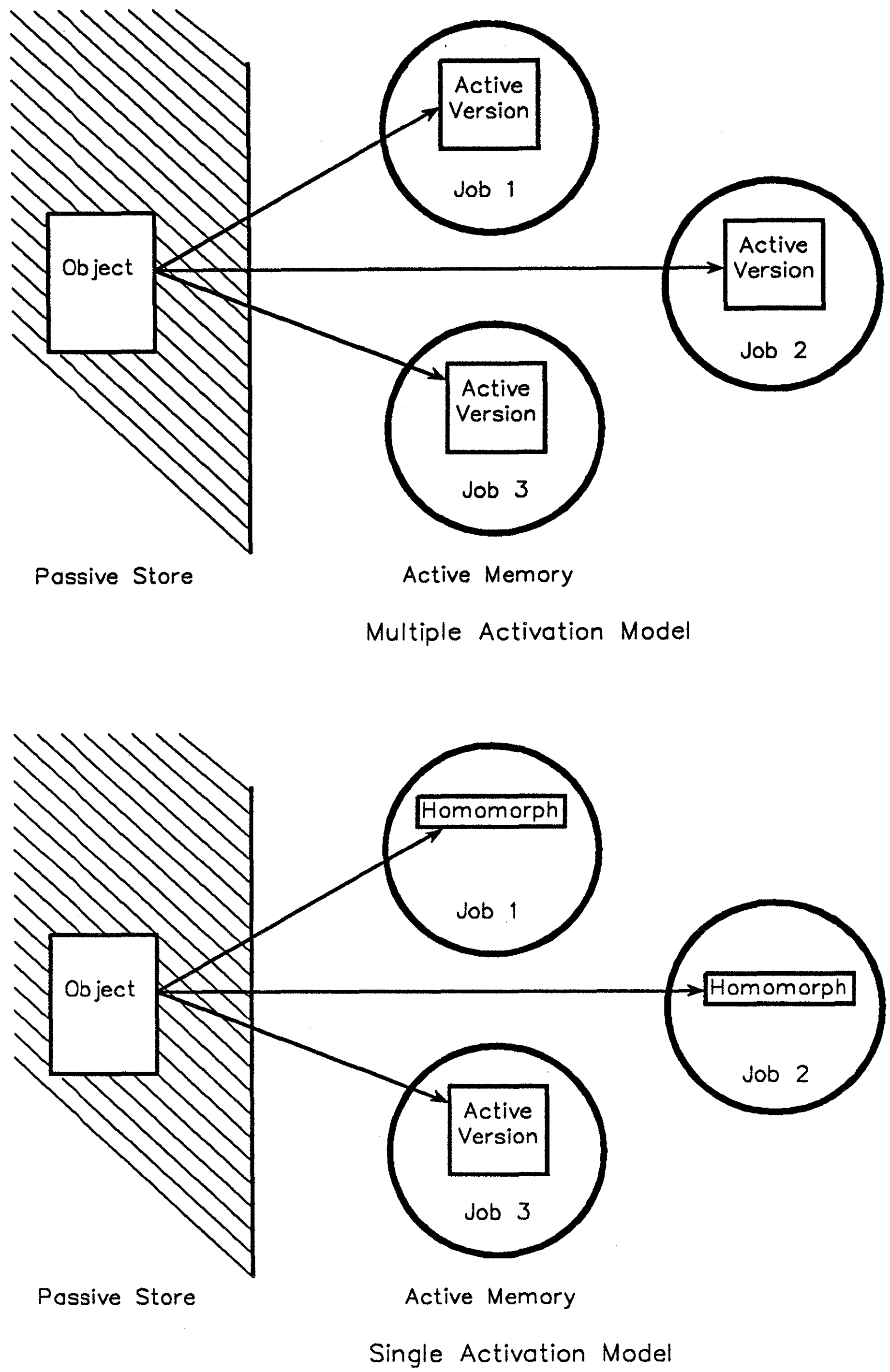

Figure VIII-1-4. Single and Multiple Activation Model

Distributed services can be built along the lines of either activation model. Very little knowledge of distribution is needed in order to build a multiple activation distributed service. BiiN™ Operating System distributed services take care of the distribution part transparently in this case. Building a distributed service along the lines of the single activation model is more complicated and requires knowledge of the mechanisms of distribution and interjob communication. In the following section we shall present the model of a single activation distributed service.

VIII-1.6 Single Activation Distributed Services

There are two ways a distributed service can be set up, as a *regional* or as a *universal* distributed service. In both cases the service contains *agents* and *servers*. Requests to the service are directed to an agent. The agent forwards the request to a server which executes it and returns the results to the agent. A universal service has servers and agents on every node of the system. An example of a universal service is the filing service. A regional service has an agent on every node but servers on only a few or even a single node. An example of a regional service is a print service with a printer that is mounted on one particular node, but accepts print jobs on any node.

In a regional service an agent knows the address of at least one server. It does not have to know the address of the server that will actually execute the request; if it directs the request to another server the request will be forwarded until it reaches its destination.

A distributed type manager is also a distributed service. The difference between a type manager and a distributed service in general is that the type manager has representation rights to its objects. It can therefore distinguish between homomorphs and real active versions. This simplifies the model somewhat: There is no need for a strict two level implementation according to the client/server model. In one job the same code can act as the client, in another as the server. The code decides what role it assumes depending on whether it was handed a homomorph or the real active version. If it is handed a homomorph it recognizes that it executes outside the home job. In this case it will act as an agent and forward the request to the server. If it is handed the real active version, that means that it executes inside the home job. In that case it assumes the role of the server and executes the requests directly.

VIII-1.7 Protection in a Distributed System

Security issues constitute a considerable problem in an open network architecture. In some sense, communications over such a network are similar to radio broadcasts; it is impossible to prevent somebody from broadcasting or from listening to certain broadcasts. If you want to protect broadcasted messages you will have to encrypt them.

The only security mechanisms in effect at the transport level are those that protect TSAPs. Three rights are defined for TSAPs: *Receive*, *Send* and *Control*. Receive rights are necessary to receive messages through a TSAP. Send rights are required to send messages through a TSAP. Control rights are needed to destroy or configure a TSAP.

This protection mechanism does not prevent you from using either datagrams or virtual circuits to send messages to a TSAP on another node or even on your node if you have the TSAP's address. Validation of messages and authentication of the sender is entirely a a high level concern. There are two sides to this problem; on one side data in transit should be protected from unauthorized use. On the other side a distributed service's private ADs have to be protected from unauthorized use but at the same time be available to all instances of the service.

Encryption protects data in transit. An application that transmits sensitive data should therefore encrypt that data. There are two solutions to the problem of protecting private ADs.

(Encrypting the data to be transmitted but not protecting private ADs would be like locking the door to one's house but leaving the keys in the lock.) A distributed service can set up its own ID (identical to a user ID). Private ADs can then be stored under well-known pathnames but with an authority list that excludes all IDs but the service's ID. Another solution to the problem is to store the private ADs inside the code of the service, more specifically inside the service's static data object. This simple solution has the disadvantage that all instances of the service have to communicate when one of the private ADs changes.

Remote procedure calls provide authentication and validation services. They also protect data in transit and convert active ADs to their passive version. (An AD still has to be passivated before being transmitted in an RPC -- using an AD on another node if that AD has not been passivated before may have unexpected results.)

When using datagrams or virtual circuits the user has to provide those services himself.

VIII-1.8 Transparently Distributed Services

The BiiN™ Operating System provides a number of transparently distributed services. With the help of these services a user can take full advantage of a distributed system. They can also be used as tools to build distributed applications. Examples of these services are the filing service, the object service, the concurrent programming service, and the transaction service.

All of the BiiN™ Operating System's distributed services provide transparent access to an entire distributed system's resources. The programmer need not be aware of any of the physical peculiarities of the system.

In the following we shall list some of the most important distributed services:

VIII-1.8.1 Passive Store

`Passive_Store_Mgt` maintains a system-wide permanent storage. Objects may be stored on volume sets anywhere in the system and can be retrieved from anywhere. Passive store also maintains unique names for all its stored objects. Those names are called *unique identifiers* (UIDs). UIDs are unique not only on one distributed system but on all distributed BiiN™ systems for all times. A volume set may thus be taken from one node in a system to another or even from one distributed system to another. Objects stored on the volume set are always uniquely identified.

VIII-1.8.2 Directories

`Directory_Mgt` maintains a system-wide directory structure. Directories implement symbolic naming for stored and for active objects. Often `Directory_Mgt` and `Passive_Store_Mgt` will cooperate closely, the former providing the naming mechanism and the latter the actual storing of objects.

However, `Directory_Mgt` may stand on its own: Directory entries can reference any object, active objects as well as passive objects. And while most directories are stored, there are also active-only directories.

The directory structure on each node replicates to a certain extent the entire naming domain the node belongs to. (Certain local aliases may exist on one node, so the directory trees on two nodes are not identical, but their structure is very similar.) The directory structure is not a simple tree structure: Branches are interconnected and entries may reference backwards in the tree. Thus many different pathames may reference the same object.

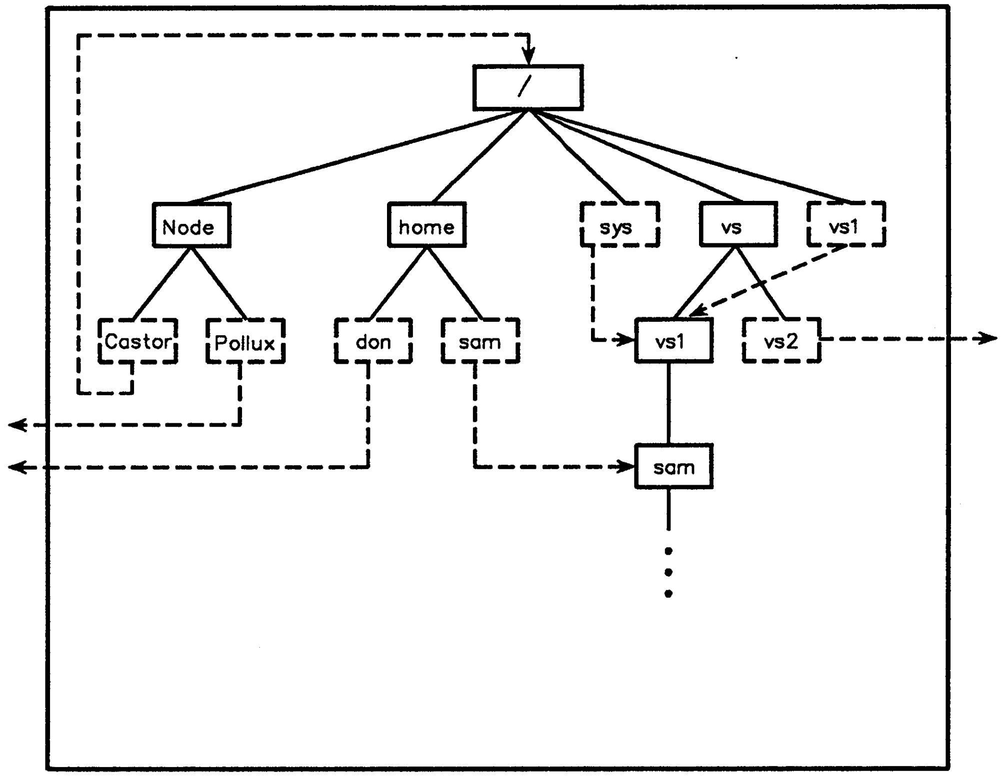

Figure VIII-1-5. Partial View of a Node's Directory Structure

Figure VIII-1-5 shows a partial view of a node's directory structure. (Solid boxes are master entries and dashed boxes represent alias entries.) In particular it illustrates that more than one pathname may reference the same object. For example, `/node/Castor/sys/sam`, `/home/sam`, and `/vs/vs1/sam` all reference Sam's home directory. By the same token `/home/don` references Don's home directory which lives on a different node. This shows that objects with two similar pathnames (`/home/sam` and `/home/don`) do not have to be physically close to each other.

VIII-1.8.3 IDs

IDs are associated with users. User IDs control access to stored objects and facilitate setting up individualized user environments. A user can be granted access to a distributed system by the system administrator. At that time the system administrator will create a user ID. A user ID grants access to an entire distributed system, not a particular node. Privileges, such as store rights for directories, are granted on a per naming domain basis.

Every process that a user starts and every object that the user stores carries the user ID. IDs are maintained in the clearinghouse's `id` environment.

Very similar to user IDs are *subsystem IDs* A subsystem ID identifies a subsystem which comprises a collection of domains that share the same stack.

There are other IDs, namely *node IDs*, *volume set IDs*, and *distributed service IDs*. All these IDs play important roles in a distributed system. Node IDs are derived from a hardware module inside a node. They are used in the node environment to map nodes to network addresses.

Volume set IDs uniquely identify volume sets. Together with a time stamp they are incorporated into unique identifiers for objects (passive ADs). Volume set IDs of volume sets mounted locally are cached to allow access to locally stored objects when there is no direct access to the clearing house.

In summary IDs are used whenever certain entities such as users or nodes are to be uniquely identified within a distributed system.

VIII-1.8.4 Files

Files are among the most important data structures in the BiiN™ architecture. Filing is a distributed service. This means that any file in the system is available anywhere in the system.

Files are global *single activation* objects; files are activated in only one place, namely at their home node. All jobs that use a particular file communicate with the home node when updating the file or reading from the file. Commonly files are large objects. Therefore it makes sense to bring the operation to the data as opposed to bringing the data to the operation.

VIII-1.8.5 Data Integrity, Synchronization, and Transactions

Data integrity and synchronization across job and node boundaries can be ensured by using transactions. Transactions make operations atomic thus preventing partial completion of operations: Operations included in a transaction either complete successfully or have no effect. Not all operations can be included in a transaction; certain operations are simply irrevocable. Printing is an example: once a page is printed it cannot be made to disappear.

Transactions extend across node boundaries whenever transaction-oriented, distributed BiiN™ Operating System service calls are included in a transaction. Transactions also serve to synchronize access to stored objects; a transaction can reserve an object on its behalf. Then no other transaction can reserve or access the object until the first transaction releases it. Transactions also have a built-in blocking protocol: One transaction can wait for another transaction only if the other transaction is older. (This ordering prevents a circular deadlock situation.)

VIII-1.9 Summary

Reading this chapter, you have learned that

- distribution makes a collection of BiiN™ nodes connected together, appear as one machine.
- a distributed system is a flexible structure; nodes may be added and removed as the system runs. In particular, distributed services do not depend on the structure of the network that connects the nodes in the system.
- logical organization hides physical organization.
- nodes share a common pool of resources, such as I/O devices, and permanent storage.
- distribution is transparent from the casual user's point of view.

BUILDING A DISTRIBUTED TYPE MANAGER 2

Contents

This chapter describes how to build a distributed type manager. It focuses on the peculiarities of the *regional service* model. Other features needed for the program, such as transactions, passivating objects, and synchronization are described in chapter VII-6. The basic concepts of the type manager model are treated in chapter VII-3.

Three packages and two initialization procedures are described in this chapter, `Account_Mgt_Ex`, `Distr_Acct_Call_Stub_Ex`, `Distr_Acct_Server_Stub_Ex`, `Distr_acct_init_ex`, and `Distr_acct_home_job_ex`. These packages will be refered to briefly as *core, call stub, server stub, initialization*, and *home job initialization.* All packages and the initialization procedures can be found in Appendix X-A.

VIII-2.1 Concepts

The type manager described here manages *local* objects on a distributed system that may consist of any number of nodes grouped into any number of naming domains. Active versions of local objects are confined to a single job, and each job activating the object receives its own active version (Some of the active versions may be ''ersatz'' versions). All processes of one job share the job's active version. (*Global* objects have only one active version per node shared by all jobs on that node.)

According to the single activation model, the object's representation is activated in one *home job*. All operations and all synchronization are handled by the home job. Other jobs receive token active versions called *homomorph* and do not operate on the object directly -- they forward all requests to the home job.

As an alternative, a type manager may use the *multiple activation model*: In the multiple activation model every job receives an active version. The multiple activation model is usually simpler to implement, but updating the passive version from multiple active versions has to be carefully coordinated. One can say that the multiple activation model brings the object to the operation, while the single activation model brings the operation to the object: For large objects, such as files for example, the single activation model is more efficient.

The node where the objects are managed is called the *home node*. Any node can be the home node.

The example described manages simple accounts that contain a `long_integer` balance. Accounts can be stored in directories or inside other objects anywhere on the system. When creating an account the application supplies a pathname or an object where the account is to be stored. In order to minimize network traffic it is advisable to store accounts on volume sets mounted at the home node -- the type manager does not enforce this, however. Independently of where accounts are stored they are accessible from any node of the system.

Communications between the home job and any other jobs are implemented by means of remote procedure calls. For more details on the general principle of distribution and RPCs refer to chapter VIII-1.

The type manager provides the following calls:

`Is_account` Checks whether an AD references an account.

`Create_account`
Creates an account and stores it inside an object supplied by the caller.

`Create_stored_account`
Creates an account and stores it with a pathname supplied by the caller.

`Get_balance` Returns an account's balance.

`Change_balance`
Changes an accounts balance and returns the new balance.

`Transfer` Transfers an amount between accounts.

`Destroy_account.`
Destroys an account.

VIII-2.1.1 Homomorphs and Active Versions

The type manager creates a template that is activated in place of the active version in all jobs but the home job. The template does not have to have the same type as the object it will stand for. The template merely represents a bit pattern that is copied into active memory and become the homomorph. Only the type manager using the representation rights can distinguish between homomorph and active version. The type manager can use the homomorph to store information related to a calling job. Such information can be statistical, for example frequency of calls, or use of resources.

VIII-2.1.2 The Remote Call

A call to the type manager involves two jobs, the calling job and the type manager's *server job*. The server job is also the home job. The two jobs may live on a single node or on two separate nodes.

Figure VIII-2-1 illustrates the general model of a distributed service implemented with RPCs.

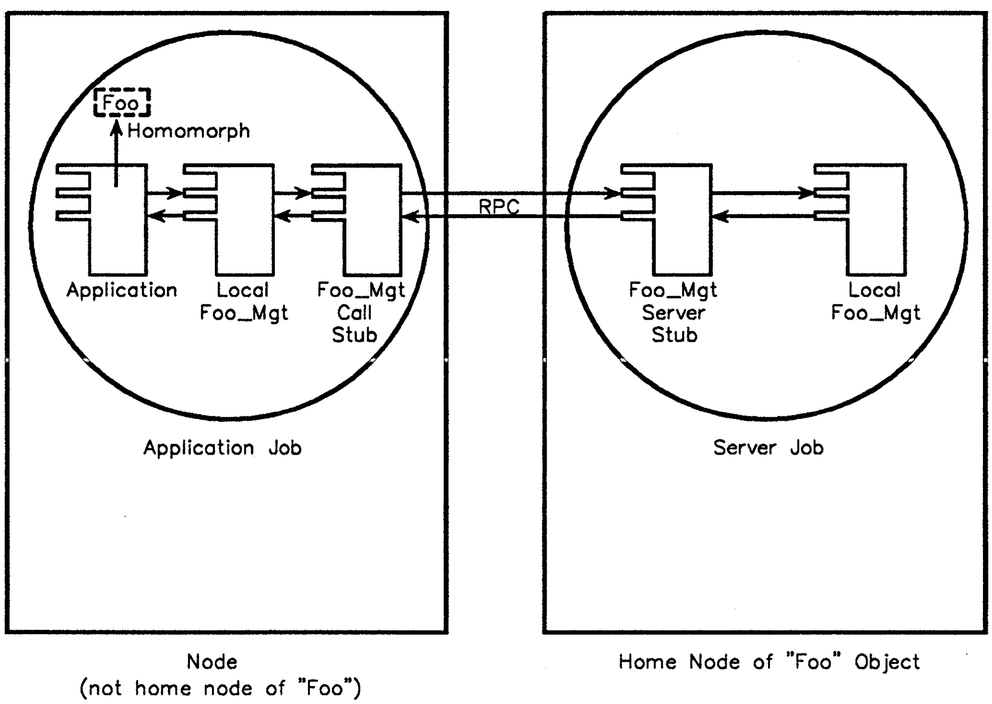

Figure VIII-2-1. General Model of Communication Using RPCs

A user program in the calling job holds an AD to the object called FOO. The calling job is not the home job of FOO objects and the AD points to a homomorph. The user program calls the local instance of FOO_Mgt, the type manager for FOO objects. FOO_Mgt recognizes from the homomorph that the job is not the home job and forwards the call to its call stub. The call stub packs the parameters into a message buffer and issues an RPC to the server. The initial program in the server is FOO_Mgt's server stub which calls the local instance of FOO_Mgt. FOO_Mgt performs the requested operation and the result is returned.

This is how the general model maps to the special case described here: Account_Mgt_Ex acts as the type manager's front end. It corresponds to Local Foo_Mgt in the picture. Applications that want to use the type manager call this package. Thus the distributed implementation looks identical from the outside to the other implementations of the account manager described in Chapters VII-3 and VII-6. All communication between different instances of the type manager on different nodes happens behind the scenes, namely in the call stub, Distr_Acct_Call_Stub_Ex, and the server stub, Distr_Acct_Server_Stub_Ex.

The actual work of the type manager is done by Account_Mgt_Ex in the home job. This package distinguishes between objects and their homomorphs. When it encounters a real object its operations are identical to the ones of the package described in Chapter VII-6 except for the semaphore synchronization mechanism. (This happens in the home job.) When it encounters a homomorph it hands off the call to the call stub that takes care of the remote calling mechanism. (This happens in an application job.) Thus the remote calling syntax is not part of the type manager's core and can be easily changed without affecting the type manager.

VIII-2.1.3 Synchronizing Access

The single activation model centralizes synchronization in the home job. Multiple simultaneous requests may be serviced by concurrent processes inside the home job. Processes in the home job share the active version of an account. Access to the active version is synchronized by *semaphores*. Semaphore locking relies on voluntary compliance of all processes. Processes that operate on an object have to call P before touching the object. This will block the calling process if another process has locked the semaphore previously. However, nothing prevents a process from circumventing the semaphore mechanism altogether.

No provisions are made to synchronize access to passive versions since according to the model of this distributed service there is never more than one active version from which the passive version can be updated.

As with all locking mechanisms there is a problem of circular waiting. Transaction come with a built-in blocking protocol that avoids this. For semaphores the problem can be solved by enclosing all semaphores within transactions to use the transaction timeout to break any circular waiting pattern.

VIII-2.2 Techniques

Packages Used:

`Access_Mgt` Interface for checking and changing rights in access descriptors.

`Attribute_Mgt` Provides a way to define general-purpose operations supported by multiple object types or objects, with different type-specific or object-specific implementations.

`Authority_List_Mgt`
Provides Calls to manage authority lists and to evaluate a caller's access rights to objects protected by authority lists.

`Directory_Mgt` Manages directories and directory entries.

`Identification_Mgt`
Provides operations to manage IDs and ID lists.

`Object_Mgt` Provides basic calls for object allocation, typing, and storage management. Defines access rights in ADs.

`Passive_Store_Mgt`
Provides a distributed object filing system.

`RPC_Call_Support.Remote_call`
Calls a service that may be at another node.

`Semaphore_Mgt.P`
Enters / locks / waits at a semaphore.

`Semaphore_Mgt.V`
Unlocks / leaves / signals a semaphore.

`Transaction_Mgt`
Provides *transactions* used to group a series of related changes to objects so that either all the changes succeed or all are rolled back.

`User_Mgt` Provides calls to manage a user's protection set and user profile.

VIII-2.2.1 Defining The Representation of The Object

In addition to other contents the type manager's objects hold two fields: A locking area and an `is_homomorph` boolean. The locking area is needed for semaphore locking and the `is_homomorph` field allows the type manager to distinguish homomorphs from active versions. The example from the core shows the account layout which contains the `long_integer` balance plus those two fields:

```
type account_rep_object is
  -- Representation of an account.
  record
    lock:  Semaphore_Mgt.semaphore_AD;
      -- Locking area
    is_homomorph:  boolean;
      -- If false identifies the object
      -- as the active version; if true
      -- as a homomorph.
    balance: Long_Integer_Defs.long_integer;
      -- Starting balance.
  end record;
```

The locking area is null in the passive version but is filled in with an AD to a semaphore when the object is activated.

The object layout is specified with an address clause. This is necessary since the type manager relies on the layout of the object in memory: Record layout in memory may vary from compiler version to compiler version.

```
FOR account_rep_object USE
record AT mod 32;
  lock          at  0  range  0 .. 31;
  is_homomorph  at  4  range  0 ..  7;
  balance       at  8  range  0 .. 63;
end record;
```

VIII-2.2.2 Defining the Homomorph Template

The homomorph template acts as a bit pattern that is copied into active memory in place of an active version. In the simplest case the template is defined with `is_homomorph` set to true while in the active version `is_homomorph` is false. Other information can be stored in the template. In particular, the type manager can use the template to store resource or statistical information pertaining to the calling job. The following example is from the initialization procedure `Distr_acct_init_ex`. (This procedure can be found in its entirety in Appendix X-A. In our example only the `is_homomorph` field is used. The other fields are initialized to null.

```
type template is
  record
    dummy_word0:    System.untyped_word;
    is_homomorph:   boolean;
    dummy_word1:    System.untyped_word;
    dummy_word2:    System.untyped_word;
  end record;

  FOR template USE
  record AT mod 32;
    dummy_word0   at  0  range  0 .. 31;
    is_homomorph  at  4  range  0 .. 7;
    dummy_word1   at  8  range  0 .. 31;
    dummy_word2   at 12  range  0 .. 31;
  end record;

type homomorph_AD is access template;
  pragma access_kind(homomorph_AD, AD);

homomorph:  homomorph_AD;
. . .
-- 2. Allocate and initialize homomorph template:
--
homomorph := new template'(
    dummy_word0  => System.null_word,
    is_homomorph => true,
    dummy_word1  => System.null_word,
    dummy_word2  => System.null_word);
```

Note that template does not even have the same type as the object proper.

VIII-2.2.3 Setting the Passive Store Attribute

In order for `Passive_Store_Mgt` to transparently substitute a homomorph for active versions in all jobs but the home job, the `homomorph` field in the `PSM_attributes_object` has to be non-null. If the field is not null `Passive_Store_Mgt` uses the AD contained in that field as a reference to a template to substitute for the object. The following excerpt from the initialization shows how the passive store attribute defined and how it is initialized:

```
passive_store_impl:
    Passive_Store_Mgt.PSM_attributes_AD;
  -- Implementation of passive store attribute
  -- for accounts.
. . .
-- 1. Allocate new passive store attribute implementation:
--
passive_store_impl := new
    Passive_Store_Mgt.PSM_attributes_object;
. . .

-- 3. Initialize passive store attribute implementation:
--
passive_store_impl.homomorph := Untyped_from_homomorph(homomorph);

passive_store_impl.reset :=
    Refuse_reset_active_version_Ex.
        Refuse_reset_active_version'subprogram_value;

passive_store_impl.copy_permitted := false;

passive_store_impl.locking_area_start := 0;
passive_store_impl.locking_area_end := 0;
  -- Area in account where semaphore AD will be
  -- stored when account is activated.
```

The `PSM_attributes_object` also specifies where the locking area is and that accounts cannot be copied.

VIII-2.2.4 Defining Buffers for Remote Procedure Calls

Buffers are necessary for both parameters and results in remote procedure calls. The following example from the server stub defines one buffer type for both parameters and results.

```
type buffer is
  -- Buffer used for remote calls.
  record
    first_word:   System.untyped_word;
    second_word:  System.untyped_word;
    amount:       Long_Integer_Defs.long_integer;
  end record;
```

The buffer has room for two ADs and one `long_integer`. This is the maximum transmitted in one single call. (`Transfer`). Note again that an address clause is used to fix the layout of the buffer in memory:

```
FOR buffer USE
  record AT mod 32;
    first_word   at  0  range  0 .. 31;
    second_word  at  4  range  0 .. 31;
    amount       at  8  range  0 .. 63;
  end record;
```

VIII-2.2.5 The `Is_` Call

Calls Used:

`Object_Mgt.Retrieve_TDO`
Returns an object's type.

No inter-job communication is necessary for the `Is` call: The object itself is not involved in the call at all: The type manager only retrieves a TDO and compares it to its own TDO. For this reason the the core does the work directly as can be seen in the following example:

```
return obj /= System.null_word
    and then
    Object_Mgt.Retrieve_TDO(obj) = account_TDO;
```

VIII-2.2.6 The The `Create_` Calls

Calls Used:

`Transaction_Mgt.Get_default_transaction`
Returns the transaction on top of the transaction stack.

`Transaction_Mgt.Start_transaction`
Starts a transaction and pushes is it on the stack.

`Transaction_Mgt.Commit_transaction`
Commits a transaction.

`Transaction_Mgt.Abort_transaction`
Aborts a transaction.

The type manager uses the `is_homomorph` field to distinguish between the home job and any other job. This method fails with the `Create_` calls since there is neither a homomorph nor an active version to check before the object has been created. (Remember that `is_homomorph` is false in the home job and true in all other jobs.)

For this reason any job can create objects. This means that in both `Create_` calls the core does the operation directly. In order to prevent multiple active versions the new object is deallocated as soon as it has been created and passivated. The three steps, *Allocate*, *Passivate* and *Deallocate* are enclosed in a transaction. Thus the `Create_` calls cannot succede partially leaving unwanted active versions.

The following excerpt from the core shows these essential part of the `Create_account` call:

```
      if Transaction_Mgt.Get_default_transaction =
          null then
        Transaction_Mgt.Start_transaction;
        trans := true;
      end if;


      begin
        -- This block controls the scope of
        -- the exception handler.

        -- 5. Create the master AD:
        --
        Directory_Mgt.Store(
            name   => master,
            object => account_untyped,
            aut    => authority);

        -- 6. Passivate the representation of the account:
        --
        Passive_Store_Mgt.Update(account_rep_untyped);

        -- 7. Deallocate the active version of the
        --    account:
        --
        Object_Mgt.Deallocate(account_rep_untyped);

        -- 8. Commit any local transaction.
        --
        if trans then
          Transaction_Mgt.Commit_transaction;
        end if;

      exception

        -- 9. If an exception occurs, abort any local
        --    transaction, deallocate the account and
        --    reraise the exception:
        --
        when others =>
          if trans then
            Transaction_Mgt.Abort_transaction;
          end if;
          Object_Mgt.Deallocate(account_rep_untyped);
          RAISE;

      end;
```

The type manager provides a second `Create_` call named `Create_stored_account`. While the `Create_account` call simply allocates a new account, the `Create_stored_account` also stores the account with a pathname supplied by the caller. The calling mechanism is identical to the `Create_account` call and the operation proper in the core is identical to the one described in Chapter VII-6.

VIII-2.2.7 Implementing Calls that Require Remote Calls

Except for the three calls discussed in the previous sections, namely `Is_account`, `Create_account`, and `Create_stored_account`, all calls of the type manager require remote calls. The remote call has the same calling syntax for jobs on one node and for jobs on different nodes. When a remote call is needed the core hands it off to the call stub that takes care of it.

VIII-2.2.7.1 Recognizing the Home Job

The `is_homomorph` field is used to recognize the home job. In the home job the type manager will see `is_homomorph` as false, in any other job as true:

```
if account_rep.is_homomorph then

  -- 2. We have a homomorph:
  --
. . .
else

  -- 3. We are in the home job for accounts:
  --
. . .
end if;
```

When `is_homomorph` is true a remote procedure call has to be made and the core hands the call off to the call stub. When `is_homomorph` is false the operation can be done directly.

VIII-2.2.7.2 Making the Remote Procedure Call

Calls Used:

`RPC_Call_Support.Remote_Call`
Makes an RPC to an RPC service.

A remote procedure call is a means of communication between two jobs. All information passed between the jobs is contained in buffers.

Both the caller and the callee have to agree on the format of the buffers. Once transmitted to another job a buffer is no more than a pattern of bits that has to be interpreted correctly. Two buffers are required, one for parameters and one for results. This is shown in the following example from the call stub:

```
parameters, results:  Distr_Acct_Server_Stub_Ex.buffer;
```

For the type declaration of `buffer` refer to section VIII-2.2.4. Before the call the calling job packs parameters into the buffer and after the call results are unpacked from the results buffer:

```
parameters := Distr_Acct_Server_Stub_Ex.buffer'(
    first_word  => account_untyped,
    second_word => System.null_word,
      -- irrelevant
    amount      => Long_Integer_Defs.zero);
      -- irrelevant
. . .
current_balance := results.amount;
```

The layout of the buffer is designed for maximum required size. Not all slots are needed in all calls.

When making a remote call the calling job specifies the service to be called. This directs the call to a server job where the service is currently registered. Optionally a node ID can be specified in the call. This will direct the call to the server on the specified node. This option can be used when multiple servers exist and one in particular is to be chosen.

The calling job also specifies an ordinal value called `target_proc`. The main package's calls are assigned an ordinal value and depending on the value of `traget_proc` in the call the associated procedure or function in the main package is called.

In the case of our example the assignments are as follows:

0	Used to initialize the server job.
1	`Get_balance.`
2	`Change_balance.`
3	`Transfer.`
4	`Destroy_account.`

Note that `Is_account`, `Create_account`, and `Create_named_acount` are not assigned an ordinal value. These functions are always performed locally and do not require a remote call.

The addresses and sizes of the buffers are also specified, and a boolean parameter is used to indicate that ADs are being transmitted. ADs have to be converted in a remote call. Indicating that no ADs are present speeds up the call.

The following example shows the syntax of the remote call:

```
length := RPC_Call_Support.Remote_call(
    service        => service,
    target_proc    => 1,
    param_buf      => parameters'address,
    param_length   => parameters'size,
    ADs_present    => true,
    results_buf    => results'address,
    results_length => results'size);
```

As you can see from the above assignments this remote call will result in `Get_balance` being called by the server. The variable `length` contains the actual length of the results buffer. This is useful when the result buffer's length varies. The variable is not used here since the results buffer in this example has a fixed length. In order to see where `service` comes from refer to section VIII-2.2.9.3.

VIII-2.2.7.3 The Server Stub

Calls Used:

`RPC_Mgt.Server_stub`
: Template for a stub procedure to be called by the server.

When the server is called it executes an initial procedure called the ***server stub***. The procedure declaration of the server stub matches a template, namely `RPC_Mgt.Server_stub`. The type manager provides the implementation of the template. The declaration looks like this:

```
procedure server_stub(
    target_proc:              System.short_ordinal;
    version:                  System.ordinal;
    param_buf:                System.address;
    param_length:             System.ordinal;
    results_buf:              System.address;
    results_length:    in out System.ordinal;
    ADs_returned:         out boolean)
```

Depending on the value of `target_proc` the server stub interprets the parameter buffer and makes the requested call. In the example the server stub is coded with a `case` statement:

```
case target_proc is
. . .
  when 2  => account_one_untyped := parameters.first_word;
             amount :=
                 Account_Mgt_Ex.Change_balance(
                    account =>
                        account_one,
                    amount  =>
                        parameters.amount);
             results := buffer'(
                 first_word  => System.null_word,
                   -- irrelevant.
                 second_word => System.null_word,
                   -- irrelevant.
                 amount      => amount);
             ADs_returned := false;
. . .
  when others   =>
      RAISE System_Exceptions.operation_not_supported;
end case;
```

Note that the server stub calls the core. This does not result in an infinite loop by triggering another remote call since this call takes place inside the home job. The core performs the requested operation and returns the result.

VIII-2.2.8 Synchronizing with Transactions and Semaphores

Access to account objects is centrally synchronized in the home job. In the home job multiple concurrent processes may access an account. Concurrent processes in the home job use *semaphore locking* to reserve the active version of an account. More details on synchronization and semaphore locking can be found in Chapters VI-1 and VI-2.

- Access to the passive version of an account is not synchronized since no more than one active version of an account exists. Here lies one of the advantages of the single activation model.
- Transactions are used to prevent semaphore deadlock and to protect passive versions from incomplete updates. Please note that the transaction timeout period is set when the system is configured.
- Outside the home job no synchronization is required since object representations are never touched outside the home job.

VIII-2.2.9 Initialization

This type manager is a distributed service and spans at least two jobs. Two procedures are needed to initialize the type manager, `Distr_acct_init_ex`, and `Distr_acct_home_job_ex`. Both procedures can be found in Appendix X-A.

The following three points should be considered when the service is initialized:

- Depending on how the service is set up it may or may not create a lot of network traffic. The worst possible situation arises when the type manager's image module is stored on one node, the stub on another, the home node is still another node, and accounts are stored all over the network. Objects should be stored close to the home node, ideally on the home node itself.
- The type manager model of protection can only be fully realized if the code is linked into its own separate domain. In particular, the type manager's private ADs are hidden in the static data object with the help of the BiiN™ Ada pragma `bind` at link-time. Therefore the static data object should not be accessible to any other module but the type manager.

- As part of the initialization the server is created and installed. When installing the server the caller can specify an SSO from which the server is scheduled and a cpu time limit. If those parameters are not explicitly specified (as in our example) the server is allocated from the caller's SSO and inherits the caller's time limit. For this reason the type manager should be installed from a privileged ID. Otherwise the server may experience resource exhaustion at some unexpected time.

VIII-2.2.9.1 Private ADs are Hidden in the Static Data Object.

The ADs for the TDO and the service are stored in the type manager's module, more precisely the static data object. This is necessary since these objects are created by the `Distr_acct_init_ex` procedure and stored with an authority list that includes only the developer thus making them inaccessible to the user of the type manager. They are retrieved when then type manager is linked. For this reason linking has to be done with the developer's ID. A third AD, the one for the homomorph, is stored by the `Distr_acct_init_ex` procedure in the passive store attribute.

The objects referenced by these ADs should only be created once. For example: One type is identified by exactly one TDO. There cannot be two TDOs referencing the same type. By definition two objects referencing different TDOs have different type. (If a TDO is destroyed it can of course be replaced by a new one.) By the same token there is only one distributed service, and one homomorph template. For this reason `Distr_acct_init_ex` should only be executed once on a distributed system, prior to linking the type manager. Then, after the type manager has been linked, `Distr_acct_home_job_ex` should be executed to initialize the server.

After these steps have been executed the main package can be called by an application. The following sections explain the steps in the initialization:

VIII-2.2.9.2 Creating the Server

Calls Used:

`RPC_Mgt.Create_RPC_server`
: Creates an RPC server.

`RPC_Mgt.Install_server`
: Installs an RPC server and returns an AD to the server job.

The following call creates a server on the local node:

```
server:  constant  RPC_Mgt.RPC_server_AD :=
    RPC_Mgt.Create_RPC_server;
  -- Server for accounts.

server_job:  Job_Types.job_AD;
  -- Installed server job.
. . .
-- 7. Install server:
--
server_job := RPC_Mgt.Install_RPC_server(
    server => server);
```

Four optional parameters can be specified with the call (default values are given in parentheses): A maximum (2) and a minimum (2) number of processes for the server, a maximum number of services (1) that can be registered with the server, and a naming domain with which the server will associate. (naming domain of the creating node). Note that two steps have to be taken to create the server, first it has to be created, second it has to be installed. Installing the server creates the server job. This example package should first be called by a job with unlimited resources, or an unlimited SSO should be specified in this call.

VIII-2.2.9.3 Creating and Registering the Service

Calls Used:

`RPC_Mgt.Create_RPC_service`
: Creates an RPC service and returns an AD to the service.

`RPC_Mgt.Register_RPC_service`
: Registers a service with a server. More than one service can be registered with one server.

An RPC service is transparently accessible. That means that the caller does not have to know the physical address of the server, but can specify the service and the call will be routed transparently. The service is not automatically associated with a server. In order to bind a service to a server the service has to be registered with the server. Multiple services can be registered with one service. Exactly how many is determined by the `max_services` parameter in the `RPC_Mgt.Create_RPC_Server` call. The following excerpt from the initialization shows these two calls:

```
-- 8. Create the service:
--
service := RPC_Mgt.Create_RPC_service(
   server => server);

```

When registering a service the caller specifies a stub procedure. That stub procedure matches the `RPC_Mgt.Server_stub` template. The server executes the stub procedure registered with one service when it receives a remote call from that service.

VIII-2.2.9.4 Setting Up the Home Job

Calls Used:

`Passive_Store_Mgt.Set_home_job`
: Establishes the calling job as home job for objects of one type. Undoes the effect of any previous call by another job.

Before the service can be called the server has to become the home job for account objects. This is achieved by executing the `Distr_acct_home_job_ex` procedure. The following excerpt shows this procedure in its entirety:

```
begin
  -- Set up server as home job
  --     by calling procedure ''0'':
  --
  parameters := Distr_Acct_Server_Stub_Ex.buffer'(
  first_word  => account_TDO_untyped,
    -- account TDO
  second_word => System.null_word, -- Irrelevant.
      amount      => Long_Integer_Defs.zero);
    -- Irrelevant.

  length := RPC_Call_Support.Remote_call(
      service        => service,
      target_proc    => 0,
        -- Server will call Passive_Store_Mgt.Set_home_job.
      param_buf      => parameters'address,
      param_length   => parameters'size,
      ADs_present    => true,
      results_buf    => results'address,
      results_length => results'size);

end Distr_Acct_Home_Job_Ex;
```

This procedure makes a remote call specifying 0 as the target procedure. In turn, the server stub which is running in the server job calls `Passive_Store_Mgt.Set_home_job` when 0 is specified as the target procedure:

```
     case target_proc is
       when 0  => account_TDO_untyped := parameters.first_word;
                  Passive_Store_Mgt.Set_home_job(
                      tdo => account_TDO);
                  ADs_returned := false;
 . . .
     end case;
```

Note that the `Passive_Store_Mgt.Set_home_job` procedure has to call and cannot call `Set_home_job` directly since only the server executes exclusively in the server job.

VIII-2.3 Summary

From this chapter you should have learned how to build a distributed type manager. The example described has the following properties.

- The type manager acts as a distributed service.
- Objects are managed in one home job.
- Local instances of the service communicate with the home job by remote procedure calls.

More specifically you should have learned how to

- set up the object's representation including a locking area and an `is_homomorph` field.
- initialize the passive store attribute to implement the single activation model.
- define a template that is activated instead of the object's active version in all jobs but the home job.
- define buffers for remote calls.
- create and install the server.
- create and register the service.

- define the call stub.
- recognize a homomorph.
- pack and unpack buffers.
- make remote calls.

Part IX
Device Services

This part of the *BiiN™/OS Guide* provides information about device drivers and device managers. This part contains one chapter:

Understanding Device Managers and Device Drivers
Describes the low-level I/O model and general architecture of device managers and drivers.

Device Services contains the following services and packages:

Device driver service:
```
CP_IO_Defs
CP_Mgt
CP_Resources
DD_Support
Handling_Support
Interrupt_Handling_Support
IO_Messages_Defs
IO_Messages_Ops
Region_3_Support
SCSI_Bus_Dependent_Defs
SCSI_Record_Defs
```

shared queue service:
```
Cluster_Service
IO_Shared_Queues
```

asynchronous communication service:
```
Async_Defs
```

mass storage service:
```
Bus_Independent_Disk_Defs
Bus_Independent_Streamer_Defs
Bus_Independent_Tape_Defs
Mass_Store_Reply_Codes
MS_Configuration_Defs
```

SCSI service:
```
CP_SCSI_Defs
CP_SCSI_Mgt
SCSI_Bus_Dependent_Defs
```

subnet service:
```
Carrier_Mgt
Subnet_CL_AM
Subnet_CO_AM
Subnet_Defs
Trace_Defs
Trace_Support
```

HDLC service:
```
HDLC_Mgt
```

LAN service:
```
CSMA_CD_Defs
Ethernet_LAN_Mgt
IEEE8023_LAN_Mgt
```

Part IX Overview

UNDERSTANDING DEVICE MANAGERS AND DEVICE DRIVERS 1

Contents

This chapter describes device manager and device driver architectures.

Packages Used:

`IO_Messages_Defs`
: Defines the I/O messages mechanism interface.

`IO_Messages_Ops`
: Provides driver-independent I/O message calls for device drivers.

`Cluster_Service`
: Manages cluster servers.

`IO_Shared_Queues`
: Defines the shared queues I/O mechanism.

`Port_Mgt` Provides fast interprocess communication within a job.

`CP_Mgt` This package defines the types used in communicating with a Channel Processor (CP). This includes the format of various data structures used by a Channel Processor. Furthermore, the Send_to_CP operation is defined here. It forwards an I/O message to a Channel Processor for service.

`DD_Support` Supports device drivers.

`Interrupt_Handling_Support`
: Manages interrupt handlers.

`Handling_Support`
: Provides calls to save and restore global registers.

`Region_3_Support`
: Provides a call for installing macrocode in Region 3.

`Unsafe_Object_Mgt`
: Provides special object allocation and deallocation calls.

`Countable_Object_Mgt`
: Supports type managers of countable global objects.

The relationship between an application, a device manager, device driver and a device is shown in Figure IX-1-1.

```
                               ---------------
                              / Application /
                              ---------------
                                    ^
Low-speed Applications              |           High-speed Applications
----------------------              |           -----------------------
                                    V
Character Terminal          ----------------    Filing
Print/Spool                 | Device Manager |  Volume Sets
                            ----------------    Basic Disk/Tape/Streamer
                                    ^           HDLC
                                    |           LAN
                                    V
Async                       ----------------    SCSI
HDLC                        | Device Driver  |  IPI
                            ----------------    LAN
                                    ^           HDLC
                                    |
                                    V
Low-speed                   ----------------    High-speed
CP application              |       CP       |  CP application
                            ----------------
                                    ^
                                    |
                                    V
terminal                    ----------------    disk
printer                     |     Device     |  tape
pipe                        ----------------    communications network
```

Figure IX-1-1. Device Environment

IX-1.1 Concepts

This section introduces methods, concepts and terminology necessary for understanding the role of device managers and device drivers in communicating with devices.

A typical I/O process involves the following actions:

- A device object is opened by an application using an `Open` access method call prior to sending data to a device.
- An I/O data transfer mechanism combined with a device class forms an I/O interface through which the device manager can communicate with a device driver, a CP (Channel Processor), and ultimately a device.

This chapter describes two I/O data transfer mechanisms which may be used to form an I/O interface, and describes the roles of device managers and device drivers.

IX-1.2 I/O Model

The primary elements of the I/O model are device objects, device managers and opened device objects. A *device object* is a typed object that represents a device. A single device object is associated with each device in the system. A *device manager* is a type manager that controls access to a device. Devices include files, magnetic tapes, terminals, and pipes. An *opened device object* is a typed object that represents a input/output connection between a device manager and a device. Zero, one or more opened device objects may exist for the same device. Opened device objects are analogous to I/O channels on other systems.

IX-1.2.1 Access Methods

Applications interact with device managers via access methods. An *access method* is a collection of procedures which provide a device-independent interface to perform I/O. A device object has associated with it the implementations of the access methods supported by that device. An access method is a type attribute of device objects and opened device objects.

To perform device operations, an application selects an access method and passes a device object to its `Open` operation. `Open` returns an opened device object representing an opened device channel. The opened device object is passes as a parameter when making access method calls.

A device can be simultaneously accessed by more than one access method. This is convenient, for example, when a call is made to a library function that internally uses a different access method.

IX-1.2.2 Device Managers

A *device manager* is a type manager of a specific type of device which provides a high-level interface through which an application can communicate with a device.

IX-1.2.3 Device Drivers

A *device driver* provides a device manager with access to a physical device. In the BiiN™ Series 60/80, a device driver is connected to its device through a CP. Device drivers are simplified by being connected to a CP since drivers do not need to provide such functions as handling interrupts and issuing device commands.

IX-1.2.4 Device Classes

A *device class* is a specification which defines the device-specific details necessary to access a class of device using an I/O mechanism. Device classes are used by device managers and implemented by device drivers. Device class specifications provide opening parameters (initial values for the `IO_Shared_Queues.device_state_rep`), command codes used in the Common Part of the I/O message (`IO_Messages_Defs.IO_message.command_code`), and reply codes used in the Common Part of the I/O message (`IO_Messages_Defs.IO_message.reply_record`). A device class specification used with an I/O mechanism forms a device-specific I/O interface through which device managers and device drivers may communicate on behalf of devices of the device class.

IX-1.2.5 I/O Mechanisms

The BiiN™ Operating System defines two I/O mechanisms available to device drivers:

- I/O messages
- Shared queues.

I/O messages supports high-speed, block-oriented data transfer. shared queues supports low-speed, character-oriented data transfer. These design characteristics make the I/O messages mechanism more suitable for disk I/O and network communications, and the shared queues mechanism more suitable for I/O to terminals.

Although these mechanisms are designed to provide communications between device managers and device drivers, they may also be used for device managers to communicate with other components such as other device managers. For example, a terminal might be connected to a system via a terminal concentrator on a network. The terminal device manager could use the shared queues mechanism to talk to a software component that converts the shared queues protocol to subnet message-based requests.

These mechanisms provide data transfer. The I/O messages mechanism is also used in an administrative interface.

IX-1.2.6 The I/O Messages Mechanism

The I/O messages mechanism consists of operations that device managers can call to support data transfer, including administrative functions, with high-speed, block-oriented devices such as disks, tapes and high-speed communications.

The I/O Message

An *I/O message* is an object consisting of four parts:

- Common part
- Device Driver part
- Device Manager part
- Buffer Description part.

The *Common part* of the I/O message has fields at fixed offsets that are visible to device managers, device drivers and CPs. It contains information about an I/O request including the type of request, the device involved and the number of buffers associated with the message.

The Common part contains pointers, offsets and IDs for locating the reply mechanism, the physical device, the CP, the beginning of the buffer description array and the Common part itself. Other fields identify the type of reply mechanism used, usage information about the buffer descriptions, request and reply priorities, error ID, command code and any device-specific parameters.

The *Device Driver* part follows the Common part, is variable in size depending on the device class, and is reserved for use by device drivers and CPs.

The *Device Manager* part follows the Device Driver part, is variable in size depending on the device class and is reserved for use by the Device Manager.

The *Buffer Description* part contains an array of buffer descriptions. Each buffer description contains the size and address of its buffer and use indicators. Since this array does not begin at a fixed location within the message, the Common part contains an offset field with which device drivers and device managers can locate the beginning of the array of buffer descriptions.

I/O messages may have several buffers. The buffers must be allocated in frozen memory. A device manager must not modify the buffers between the time a request is issued and the time the I/O message is returned to the device manager.

The contents of a buffer depend on the type of request and the device class associated with the I/O message. (The semantics assigned to each request are described in the device class specification/package.) Some I/O messages might not reference any buffers at all, such as a device-specific *reset* request. Other requests such as a `Read` normally require at least one buffer.

Reply Mechanism

The device manager decides the reply mechanism, interrupt reply procedure or reply port from which it will receive its returned I/O messages. The selected mechanism is specified by the values in `reply_port_or_proc` and `type_of_reply`.

The *interrupt reply procedure* is called by an interrupt handler, and performs post-processing of the serviced I/O message such as setting `error_id` and `total_returned_length`. A template for this procedure is provided via `IO_Messages_Defs.Process_IO_message`. The *reply port* mechanism is an inter-process communications mechanism on which I/O messages can be enqueued.

The interrupt reply procedure has the advantage of not causing a context switch, but does execute an interrupt handler. Thus the implementation of an interrupt reply procedure must comply with all constraints placed on interrupt handlers (see `Interrupt_Handling_Support` for a list of interrupt handler constraints). Most BiiN™ Operating System device managers use the I/O reply port mechanism.

IX-1.3 Data Transfer Via the I/O Messages Mechanism

Most systems will employ CP-connected devices because I/O via CPs is available and efficient for the more common protocols (see *BiiN™/OS Reference Manual* for a list of supported devices). Using a CP also greatly simplifies the tasks which must be performed by a device driver.

```
                 ----------------
                | Device Manager |
                 ----------------
                        |
                        |
                        V
              ------------------------
             | Device  | I/O Messages |
             | Class   |  Mechanism   |
              ------------------------
                        |
                        |
                        V
           -------------------------
          |    Channel Processor    |
           -------------------------
                     ^
                     |
                     V
                   device
```

Figure IX-1-2. Device Driver using the I/O messages Mechanism

Data transfer to a CP-connected device using the I/O messages mechanism can be done via the following steps:

1. The application calls an access method `Open` to create an opened device.

2. The device manager allocates the data buffers and buffer descriptions (optionally using DD_Support.Set_buffer_description), and fills in the following fields:

   - queuing_space
   - reply_port_or_proc
   - total_request_length
   - type_of_reply
   - reply_priority
   - io_msg
   - used_buffers, optional
   - max_buffers
   - command_code
   - buffer_descr_offset
   - device_specific_params

   The device manager may optionally allocate a pool of I/O messages by repeatedly creating I/O messages and calling DD_Support.Register_IO_message. A pool of I/O messages may be shared by several devices.

3. The device manager calls IO_Messages_Ops.Ops.Issue_request to forward the I/O message to a device for service.

4. Any time after the I/O message has been sent to the device (Step 2), the device manager calls Port_Mgt.Receive or Port_Mgt.Conditional_receive to receive the message from the reply port, if a reply port was selected as the reply mechanism. If the selected reply mechanism is an interrupt reply procedure the message receipt method is be defined by the procedure.

5. The device driver gets access to the I/O message, and fills in the following fields of the Common part of the I/O message:

   - phys_dev
   - request_priority, optional
   - cp_id
   - device_id

   The device driver also fills in the following fields defined in the Device Driver part of the I/O message required by the CP:

   - interrupt_q_addr
   - phys_buf_desc_addr

   interrupt_q_addr is the physical address of an interrupt queue head. It identifies the return path from a CP to a CPU after the message has been serviced. phys_buf_desc_addr is the physical address of the buffer description array.

   The device driver can call an access method's Get_device_info call to acquire information for some of these fields. It can also place other information in the undefined section of the Device Driver part for its own use.

The device driver must set these fields because a device manager will generally use one pool of I/O messages to issue requests for all the devices it manages. Since a device manager may manage some devices that are connected to the system by CPs and others that are directly connected, several different device drivers may service a single device manager's I/O requests. They may use the Device Driver part of the I/O messages differently. Therefore, a device driver must set all the fields in an I/O message that specify device information.

6. The device driver issues an I/O request to the CP by calling `CP_Mgt.Send_to_CP`.
7. After the CP has finished servicing the I/O request, it writes the following results in the I/O message:
   - `error_id`, if an error occurred.
   - `total_returned_length`
   - `reply_record`
8. The CP sends the I/O message to the interrupt queue specified by `interrupt_q_addr` and generates an interrupt.
9. The CPU interrupt handler which processes CP-generated interrupts, returns the I/O message to the reply mechanism specified in the I/O message (`Port_Mgt.Send` for a reply port).
10. The device manager may continue issuing requests for service, calling receive operations and logging any errors.
11. When the device manager completes and needs no further access to the device, it waits for pending I/O requests to complete (or cancels them and calls an access method's `Close` to close the opened device.
12. After the device manager has received the I/O messages from the reply mechanism (Step 3), and closed all the devices that it manages, it may optionally deregister the pool of I/O messages with the recovery agent via `DD_Support.Deregister_IO_message`.

IX-1.3.1 I/O Recovery Agent

A *recovery agent* is provided on each node by the BiiN™ Operating System. This agent detects I/O processor failures and maintains a table of existing I/O messages. Device managers keep this list current by calling `DD_Support.Register_IO_message` each time they create an I/O message, and by calling `DD_Support.Deregister_IO_message` before they deallocate an I/O message.

IX-1.4 Data Transfer Via the Shared Queues Mechanism

The shared queues I/O mechanism is designed to handle low-speed, character-oriented communications for such devices as terminals and printers. This design minimizes context switches and interrupts while maintaining satisfactory response time.

The shared queues mechanism is comprised of a cluster servers which services one or more clusters which contain up to eight pairs of input and output queues (circular buffers). This mechanism employs an input and output queue for each device. These queues are grouped into clusters. A *cluster* is a group of queues that are serviced together. A cluster represents a group of devices, typically those serviced by the same channel processor (CP) task. See Figure IX-1-3.

IX-1.5 Clusters and Cluster Servers

Clusters are configurable objects (CO) and are typically created and attached to devices during system initialization. A cluster may contain shared queues for up to eight devices. Cluster servers may service any number of clusters.

```
                      Cluster Server
                            |
     ____________________________________  . . .  ____
     |                      |                         |
  Cluster 1              Cluster 2                 Cluster n
     |                      |                         |
     ~          ------------------ . . . -----        ~
                |           |                 |
             Device 1    Device 2   . . .  Device 8
              |   |       |   |             |   |
             in  out     in  out           in  out
             queues      queues            queues
```

Figure IX-1-3. Cluster Server, Clusters and shared queues

The devices of each cluster must be of the same device class.

IX-1.5.1 Administrative Interface

The shared queues I/O mechanism is a data transfer mechanism. Each device class that uses this mechanism must also specify an administrative interface. An administrative interface contains operations which initialize queues, set device parameters, etc.

When the I/O messages mechanism is used as an administrative interface, for example, the device class specification defines device-specific command codes and reply records and is used to initialize the clusters.

IX-1.5.2 Device Driver Example

Figures IX-1-4 and IX-1-5 show how shared queues work with CPs and their relationship with an administrative interface.

```
                              ----------------
                              | Device Manager |
                              ----------------
                                  |       ^
                                  |       |
                                  V       V
              ------------------------   ----------------
              |Administrative Interface|  \ Cluster Server \
              |------------------------|  ---------------
              | Device | shared queues |         |      |...|
              | Class  |  Mechanism    |   ---------
              ------------------------    | Cluster |
                          |               ---------
                          |                    ^
                          V                    V
                         -------------------------
                         |     Channel Processor     |
                         -------------------------
                                     ^
                                     |
                                     V
                                   device
```

Figure IX-1-4. Device Driver with the Shared Queues Mechanism

IX-1.5.3 I/O Shared Queues Data Transfer Mechanism

An input and an output queue are used to support data transfer between a device manager and a low-speed device via a CP/device driver. Each queue has a read pointer and a write pointer which indicate where the next character will be read or written, flags to indicate queues needing service and semaphores to block writers when queues are full. The data transfer process consists of four distinct activities:

- **Data Transfer From the Device Manager to the Output Queue**

  The device manager writes data to the output queue.

- **Data Transfer From the Output Queue to the Device**

  The CP/device driver polls its devices' output queues, and transfers any characters to those devices.

- **Data Transfer From the Device to the Input Queue**

  The device interrupts the CP/device driver when it has characters to be returned to the device manager. The CP/device driver transfers the data to the input queue.

- **Data Transfer From the Input Queue to the Device Manager**

  The cluster server polls its clusters and calls an input handler for any input queue containing characters.

These activities are described in more detail following Figure IX-1-5.

```
 -------------
 \ application \
  -------------
        ^
        |
        V
  --------      ----------------
 | buffer |<-->| Device Manager |
  --------      ----------------

                              ---------------...------
          Output Queue       |  |  |  |  |  |   |  |  |
                              ---------------...------
                                 ^              ^    ---------
                                 |              |    \ Cluster\
                                 R              W     \ Server \
                                                       --------
                              ---------------...------
           Input Queue       |  |  |  |  |  |   |  |  |
                              ---------------...------
                              ^  ^
                              |  |
                              W  R

           ------------------
          | CP/Device Driver |
           ------------------
                   ^
                   |
                   V
                 Device
```

Figure IX-1-5. I/O shared queues Data Transfer Mechanism

1. A device manager transfers characters from an application's buffer to the output shared queue associated with the device.
2. When each write completes, the `cluster_object.new_output_flags` flag corresponding to the output queue associated with the device is set to show that this output queue is active (contains characters to be transferred to the device).
3. If the output queue fills before the device manager completes a write, `cluster_object.new_input_flags` is still set to active, and the writer blocks on `device_state_rep.block_user`. The device manager sets the boolean `device_state_rep.writer_blocked` to true.
4. The cluster server periodically checks the state of the output queue, and unblocks the writer when the contents of the output queue reach a low enough number of characters that more characters can be accepted.
5. When the number of characters remaining in the output queue is less than a `low_water_mark` (`device_state_rep.low_water_mark`), the cluster server unblocks the writer (calls `Semaphore_Mgt.V`), sets `device_state_rep.block_user` to false and calls `device_state_rep.input_handler`. This optimization technique prevents excessive blocking and context switching.

`device_state_rep.output_write_ptr` and `device_state_rep.output_read_ptr` are pointers for the output queue that indicate where to write and where to begin reading the next character. The device manager writes characters beginning at the location indicated by the write pointer, and increments the pointer by the number of characters written. Likewise, the device manager reads characters beginning at the location indicated by the read pointer and increments the pointer by the number of characters read.

The queue is empty when the read pointer is equal to the write pointer. The queue is full when the read pointer is one more than the write pointer `mod` the queue size.

Data Transfer From the Output Queue to the Device

1. A CP/device driver periodically reads `cluster_object.new_output_flags` to determine if any of its device's output queues needs to be serviced.
2. For each active device, it sets the device's output flag in `cluster_object.new_output_flags` to false and sends a character to the device starting an interrupt-driven transfer loop.

The interrupt-driven loop is initiated by the CP/device driver when it polls the output queue and finds the new output flag set. The interrupt routine sets the new output flag to false and sends a character from the output queue to the device. (The flag must be reset before the character is sent.) When the device interrupts the CP/device driver to acknowledge receipt of the character, the loop checks the output queue again for another character to be sent. This loop continues until there are no more characters in the output queue.

NOTE

Occasionally, an output queue is marked active for which the interrupt-driven output transfer loop is in progress. The CP can detect this situation because it maintains an internal flag for each device that indicates whether or not a send is in progress. If a send is in progress, the CP marks the queue as inactive and moves on to the next active output queue.

Data Transfer From the Device to the Input Queue

1. The device sends an interrupt to the CP/device driver when it has a character to send. The CP/device driver calls an interrupt handler which places the character in the input queue, and sets the new input flag to true. (The character must be sent before the flag is reset.)
2. If the CP/device driver is unable to put a character in an input queue because the queue is full, it discards the character and sets the queue's overflow boolean, `input_lost`.

The use of the pointers in the input queue is similar to the use with the output queues except that the CP/device driver writes the characters using the write pointer and the device manager reads the character using the read pointer. A CP/device driver updates the read pointer of the output queue when removing characters. A CP/device driver reads the characters at the read pointer and increments the read pointer.

Data Transfer From the Input Queue to the Device Manager

1. The cluster server periodically checks the new input flags. If an input flag is set, the cluster server calls the input handler for the device (`device_state_rep.input_handler`).

IX-1.6 Summary

- A device object is a typed object that represents a device.
- A device manager is a type manager that controls access to a device.
- An opened device object is a typed object that represents an input/output connection between a device manager and a device.
- A device class is a specification that defines the device-specific details necessary to access a member of a class of devices using an I/O mechanism.
- An access method is a collection of procedures that provide a device-independent interface to perform I/O.
- The I/O messages data transfer mechanism supports high-speed, block-oriented data transfer.
- The shared queues data transfer mechanism supports low-speed, character-oriented data transfer.
- An I/O message is an object consisting of four parts: common part, device driver part, device manager part and buffer description part.
- A recovery agent detects I/O processor failures and maintains a table of existing I/O messages.

Part X
Appendixes

The appendixes are:

Ada Examples	Contains complete listings of all examples used in this guide.
Glossary	Defines terms used in this guide.

ADA EXAMPLES A

Contents

X-A.1 Introduction

This appendix contains full listings of all the examples in the *BiiN™/OS Guide* grouped by service area.

All examples were compiled using Version V1.00.02 of the BiiN™ Ada compiler, and all compiled successfully (except where noted). Most examples are not yet tested, however.

X-A.2 Support Services

X-A.2.1 Example_Messages Package Specification

```
with Incident_Defs,
     System,
     System_Defs;

package Example_Messages is
  --
  -- Function:
  --   Define messages used by example programs.
  --
  --   A single message file is used.  All messages
  --   defined use a module ID of 0.

  msg_file_pathname:  constant System_Defs.text_AD :=
      new System_Defs.text'(
          30,30,"/examples/msg/example_messages");
    -- AD to pathname of message file, bound to
    -- "msg_obj", following.
    --
    -- *This will go away when "pragma bind" changes.*

  msg_obj:  constant System.untyped_word :=
      System.null_word;
    pragma bind(msg_obj,
                "example_messages.msg_file_pathname");
    -- Message object for incident codes in
    -- example programs, bound to above "message_file_pathname".
    --
    -- *When the resident compiler and linker are*
    -- *ready, this pragma will become:*
    -- |  pragma bind(msg_obj,
    -- |              "/examples/msg/example_messages");


  not_directory_code:
      constant Incident_Defs.incident_code :=
      (0, 1, Incident_Defs.error, msg_obj);
  --
  --*M* store :module=0 :number=1 \
  --*M*     :msg_name=not_directory_code \
  --*M*     :short = \
  --*M*     "$p1<pathname> is not a directory."

  not_exist_or_no_access_code:
      constant Incident_Defs.incident_code :=
      (0, 2, Incident_Defs.error, msg_obj);
  --
  --*M* store :module=0 :number=2 \
  --*M*     :msg_name=not_exist_or_no_access_code \
  --*M*     :short = \
  --*M*     "$p1<pathname> does not exist or does\
  --*M* not allow you access."

  no_access_code:
      constant Incident_Defs.incident_code :=
      (0, 3, Incident_Defs.error, msg_obj);
  --
  --*M* store :module=0 :number=3 \
  --*M*     :msg_name=no_access_code \
  --*M*     :short = \
  --*M*     "$p1<pathname> does not allow\
  --*M* you access."

  overwrite_query_code:
      constant Incident_Defs.incident_code :=
      (0, 4, Incident_Defs.information, msg_obj);
  --
  --*M* store :module=0 :number=4 \
  --*M*     :msg_name=overwrite_query_code \
  --*M*     :short = \
  --*M*     "$p1<pathname> exists.  Overwrite it?"
  not_overwritten_code:
      constant Incident_Defs.incident_code :=
      (0, 5, Incident_Defs.error, msg_obj);
  --
```

```
    --*M* store :module=0 :number=5 \
    --*M*      :msg_name=not_overwritten_code \
    --*M*      :short = \
    --*M*      "$p1<pathname> not overwritten."

    create_name_space_aborted_code:
        constant Incident_Defs.incident_code :=
        (0, 6, Incident_Defs.information, msg_obj);
    --
    --*M* store :module=0 :number=6 \
    --*M*      :msg_name= \
    --*M*      create_name_space_aborted_code \
    --*M*      :short = "Operation aborted.\
    --*M*  No name space was created."

    name_space_created_code:
        constant Incident_Defs.incident_code :=
        (0, 7, Incident_Defs.information, msg_obj);
    --
    --*M* store :module=0 :number=7 \
    --*M*      :msg_name=name_space_created_code \
    --*M*      :short = \
    --*M*      "Name space $p1<pathname> created."

end Example_Messages;
```

X-A.2.2 `Long_Integer_Ex` Package Specification

```
with Long_Integer_Defs;

package Long_Integer_Ex is
  --
  -- Function:
  --    Provide examples of using long integers.
  --    See the package body for detailed comments.


function Long_integer_value(
    image:  string)
  return Long_Integer_Defs.long_integer;


function Get_long_integer
  return Long_Integer_Defs.long_integer;


function Multiply_divide(
    a:  integer;
    b:  integer;
    c:  integer)
  return integer;


procedure Use_it;


  pragma external;

end Long_Integer_Ex;
```

X-A.2.3 Long_Integer_Ex Package Body

```
with Byte_Stream_AM,
     Device_Defs,
     Long_Integer_Defs,
     Process_Mgt,
     Process_Mgt_Types,
     System,
     System_Exceptions;

package body Long_Integer_Ex is
  --
  -- Function:
  --   Provide examples of using long integers.
  --
  -- History:
  --   12-02-87  Martin L. Buchanan  Initial version.


function Long_integer_value(
    image:  string)
  return Long_Integer_Defs.long_integer
  --
  -- Function:
  --   Converts a string image to a long integer.
  --
  --   The image must have the following syntax:
  --   |
  --   |  image ::= {space} [sign] digit { [_] digit }
  --   |            {space}
  --   |  space ::= ' '
  --   |  sign  ::= +|-
  --   |  digit ::= 0|1|2|3|4|5|6|7|8|9
  --
  --   After leading and trailing spaces are stripped
  --   off, the remaining part of the image cannot
  --   be longer than 31 characters.
  --
  -- Notes:
  --   Unlike "Long_Integer_Defs.Long_integer_value",
  --   this function handles strings of varying length
  --   and strings that contain trailing spaces.
  --
  -- Exceptions:
  --   System_Exceptions.bad_parameter -
  --     "image" has incorrect syntax, contains a
  --     number longer than 31 characters, or contains
  --     a number that cannot be represented as a long
  --     integer.
is
  li_string:  Long_Integer_Defs.string_integer;
    -- Fixed-length string required by
    -- "Long_Integer_Defs.long_integer_value"
    -- when converting to a long integer.
  i:  integer;
    -- Will be index of right-most non-space character
    -- in "image".
  j:  integer;
    -- Will be index of left-most non-space character
    -- in "image".
  k:  integer;
    -- Will be index of left-most character in
    -- "li_string" that is copied from "image(j..i)".
  li:  Long_Integer_Defs.long_integer;
    -- The resulting long integer to return.
begin
  -- Make "i" the index of the right-most
  -- non-space character in "image":
  --
  i := image'last;
  loop
    if i < image'first then

      -- "image" contains all spaces, or is a
      -- null string:
      --
```

```
        RAISE System_Exceptions.bad_parameter;

      else
        EXIT when image(i) /= ' ';
        i := i - 1;
      end if;
    end loop;

    -- Make "j" the index of the left-most
    -- non-space character in "image".  No check
    -- is needed for "image" being null or all
    -- spaces, as those conditions are checked
    -- above.
    --
    j := image'first;
    loop
      exit when image(j) /= ' ';
      j := j + 1;
    end loop;

    if (i - j + 1) > li_string'length then

      -- The number is longer than 31 characters
      -- after stripping off spaces:
      --
      RAISE System_Exceptions.bad_parameter;

    else

      -- "k" is the index within "li_string" of the
      -- leftmost character copied from "image".  "k" is
      -- computed to satisfy the following predicate:
      -- |  i - j = li_string'last - k
      -- This predicate simply specifies that the number
      -- of source characters copied equals the number
      -- of destination characters.
      --
      k := li_string'last + j - i;

      -- Copy the significant characters from "image" to
      -- be right-justified within "li_string":
      --
      li_string(k .. li_string'last) :=
          image(j .. i);

      -- Fill any remaining left-hand characters in
      -- "li_string" with spaces:
      --
      for m in li_string'first .. k-1 loop
        li_string(m) := ' ';
      end loop;

      -- Compute and return the long integer value:
      --
      Long_Integer_Defs.Long_integer_value(
          image  => li_string,
          number => li);         -- out.
      RETURN li;

    end if;
  end Long_integer_value;


  function Get_long_integer
    return Long_Integer_Defs.long_integer
    --
    -- Function:
    --   Gets a long integer on a single line
    --   from the calling process's standard input.
    --
    -- Notes:
    --   See "Long_integer_value" in this package
    --   for a description of the required long
    --   integer syntax and of what happens if
    --   the syntax is violated.
    --
    --   There is no check for a line that's too long.
```

```
is
  LINE_SIZE:  constant integer := 80;
    -- A line read from the standard input must
    -- be <= 80 characters.
  line:    string(1 .. LINE_SIZE);
    -- Line buffer.
  length:  integer;
    -- Number of characters actually read.
begin
  -- Read the line:
  --
  length := integer(Byte_Stream_AM.Ops.Read(
      Device_Defs.opened_device(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.standard_input)),
      line'address,
      System.ordinal(LINE_SIZE)));

  -- Strip any linefeed at the end:
  --
  if line(length) = ASCII.LF then
    length := length - 1;
  end if;

  -- Convert to a long integer and return:
  --
  return Long_integer_value(line(1..length));
end Get_long_integer;


function Multiply_divide(
    a:  integer;
    b:  integer;
    c:  integer)
  return integer
    -- (a * b) / c
  --
  -- Function:
  --   Computes and returns the product of two
  --   integers divided by a third integer, using
  --   a long integer for the intermediate result
  --   to avoid overflow.
  --
  --   This function is useful for scaling and
  --   unit conversions, to avoid overflow within
  --   the calculation when the result after the
  --   division step can still be represented as
  --   an integer.
  --
  -- Exceptions:
  --   System_Exceptions.bad_parameter -
  --     Overflow or division by zero.
is
  -- Convert all parameters to long integers:
  --
  a_long:  Long_Integer_Defs.long_integer :=
      Long_Integer_Defs.Convert_to_long_integer(a);
  b_long:  Long_Integer_Defs.long_integer :=
      Long_Integer_Defs.Convert_to_long_integer(b);
  c_long:  Long_Integer_Defs.long_integer :=
      Long_Integer_Defs.Convert_to_long_integer(c);

  -- Import long integer operators:
  --
  use Long_Integer_Defs;

begin
  return Convert_to_integer( (a_long * b_long) / c_long );
end Multiply_divide;


procedure Use_it
  --
  -- Function:
  --   Show some computations with long integers.
  --
  -- Notes:
```

```
  --   This procedure is not yet testable as it
  --   is not a command and its variables are not
  --   yet displayed.
is
  -- Import long integer operators and the
  -- "long_integer" type:
  --
  use Long_Integer_Defs;

  -- Some variables to play with:
  --
  a:  long_integer;
  b:  long_integer;
  i:  integer;

  -- Declaring a negative long integer constant,
  -- the easy way and the hard way:
  --
  negative_twenty:  constant long_integer :=
      - long_integer'(0, 20);

  another_negative_twenty:  constant long_integer :=
      (16#ffff_ffff#, 16#ffff_ffec#);
    -- Use the hard way when you want a declaration
    -- elaborated at compile-time instead of
    -- at run-time.
begin
  -- Add one to a long integer:
  --
  a := a + Long_Integer_Defs.one;

  -- Add a positive integer "i" to a long integer:
  --
  b := b + long_integer'(0, System.ordinal(i));
end Use_it;


end Long_Integer_Ex;
```

X-A.2.4 Make_menu_group_DDef_ex Procedure

```
with Data_Definition_Mgt,
     Directory_Mgt,
     Passive_Store_Mgt,
     System,
     System_Defs,
     Text_Mgt;

procedure Make_menu_group_DDef_ex
  --
  -- Function:
  --   Creates and stores a menu group DDef,
  --   containing two menus and five menu items:
  --
  --|      -----------         -----------
  --|        Menu 1             Menu 2
  --|      -----------         -----------
  --|      Menu Item 1         Menu Item 1
  --|      Menu Item 2         Menu Item 2
  --|      -----------         Menu Item 3
  --|                          -----------
  --

is

  use Data_Definition_Mgt;    -- to import enumeration types

  ddf:           Data_Definition_Mgt.DDef_AD;
  untyped_ddf:   System.untyped_word;
    FOR untyped_ddf USE AT ddf'address;

  group_node:        Data_Definition_Mgt.node_reference;
  menu_list_node:    Data_Definition_Mgt.node_reference;
  menu_node:         Data_Definition_Mgt.node_reference;
  item_list_node:    Data_Definition_Mgt.node_reference;
  item_node:         Data_Definition_Mgt.node_reference;
  dont_care_node:    Data_Definition_Mgt.node_reference;
  name:              System_Defs.text(100);
  prop_value:        Data_Definition_Mgt.property_value(100);

begin

  ddf := Data_Definition_Mgt.Create_DDef;

  -- Create menu group

  Text_Mgt.Set(name, "group_node");
  group_node := Data_Definition_Mgt.Create_node(
      DDef       => ddf,
      node_name  => name,
      root       => private_root_node);

  prop_value.simple_pv := (pv_boolean, true);
  Data_Definition_Mgt.Add_property_value(
      node_ref => group_node,
      property => pi_derive_all,
      value    => prop_value);

  prop_value.simple_pv := (pv_boolean, true);
  Data_Definition_Mgt.Add_property_value(
      node_ref => group_node,
      property => pi_import,
      value    => prop_value);

  prop_value.simple_pv := (pv_type => pv_string);
  Text_Mgt.Set(prop_value.text_value, "menu_group_t");
  Data_Definition_Mgt.Add_property_value(
      node_ref => group_node,
      property => pi_DDef_name,
      value    => prop_value);

  Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
  Data_Definition_Mgt.Add_property_value(
      node_ref => group_node,
      property => pi_DDef_name,
```

```
        value    => prop_value);

    Text_Mgt.Set(name, "menu_list");
    menu_list_node := Data_Definition_Mgt.Create_field(
        record_node => group_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_node_reference, menu_node));


    -- Create the first menu ("Menu 1"):
    --
    Text_Mgt.Set(name, "menu_node");
    menu_node := Data_Definition_Mgt.Create_node(
        DDef      => ddf,
        node_name => name,
        root      => private_root_node);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_node,
        property => pi_derive_all,
        value    => prop_value);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_node,
        property => pi_import,
        value    => prop_value);

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "menu_t");
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(name, "menu_id");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => menu_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_int4, 1));

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "Menu 1");
    Text_Mgt.Set(name, "menu_title");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => menu_node,
        node_name   => name,
        property    => pi_has_value,
        value       => prop_value.simple_pv);

    Text_Mgt.Set(name, "item_list");
    item_list_node := Data_Definition_Mgt.Create_field(
        record_node => menu_node,
        node_name   => name);


    -- Now create the menu items for menu 1:
    --

    -- Create menu item 1:
    --
    Text_Mgt.Set(name, "item_node");
    item_node := Data_Definition_Mgt.Create_node(
        DDef      => ddf,
        node_name => name,
        root      => private_root_node);

    prop_value.simple_pv := (pv_boolean, true);
```

```
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_derive_all,
    value    => prop_value);

prop_value.simple_pv := (pv_boolean, true);
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_import,
    value    => prop_value);

prop_value.simple_pv := (pv_type => pv_string);
Text_Mgt.Set(prop_value.text_value, "menu_item_t");
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_DDef_name,
    value    => prop_value);

Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_DDef_name,
    value    => prop_value);

Text_Mgt.Set(name, "item_id");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => (pv_int4, 1));

Text_Mgt.Set(name, "checked");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => (pv_boolean, true));

Text_Mgt.Set(name, "enabled");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => (pv_boolean, true));

prop_value.simple_pv := (pv_type => pv_string);
Text_Mgt.Set(prop_value.text_value, "Menu Item 1");
Text_Mgt.Set(name, "text");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => prop_value.simple_pv);

-- Add menu item 1 to menu 1:
--
prop_value.simple_pv := (pv_node_reference, item_node);
Data_Definition_Mgt.Add_property_value(
    node_ref => item_list_node,
    property => pi_has_value,
    value    => prop_value);


-- Create menu item 2 for menu 1:
--
Text_Mgt.Set(name, "item_node");
item_node := Data_Definition_Mgt.Create_node(
    DDef      => ddf,
    node_name => name,
    root      => private_root_node);

prop_value.simple_pv := (pv_boolean, true);
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_derive_all,
    value    => prop_value);

```

```
prop_value.simple_pv := (pv_boolean, true);
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_import,
    value    => prop_value);

prop_value.simple_pv := (pv_type => pv_string);
Text_Mgt.Set(prop_value.text_value, "menu_item_t");
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_DDef_name,
    value    => prop_value);

Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
Data_Definition_Mgt.Add_property_value(
    node_ref => item_node,
    property => pi_DDef_name,
    value    => prop_value);

Text_Mgt.Set(name, "item_id");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => (pv_int4, 2));

Text_Mgt.Set(name, "checked");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => (pv_boolean, false));

Text_Mgt.Set(name, "enabled");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => (pv_boolean, false));

prop_value.simple_pv := (pv_type => pv_string);
Text_Mgt.Set(prop_value.text_value, "Menu Item 2");
Text_Mgt.Set(name, "text");
dont_care_node := Data_Definition_Mgt.Create_field(
    record_node => item_node,
    node_name   => name,
    property    => pi_has_value,
    value       => prop_value.simple_pv);


-- Add menu item 2 to menu 1:
--
prop_value.simple_pv := (pv_node_reference, item_node);
Data_Definition_Mgt.Add_property_value(
    node_ref => item_list_node,
    property => pi_has_value,
    value    => prop_value);


-- Add menu 1 to the menu group:
--
prop_value.simple_pv := (pv_node_reference, menu_node);
Data_Definition_Mgt.Add_property_value(
    node_ref => menu_list_node,
    property => pi_has_value,
    value    => prop_value);

-- Create menu 2:
--
Text_Mgt.Set(name, "menu_node");
menu_node := Data_Definition_Mgt.Create_node(
    DDef      => ddf,
    node_name => name,
    root      => private_root_node);

prop_value.simple_pv := (pv_boolean, true);
Data_Definition_Mgt.Add_property_value(
```

```
        node_ref => menu_node,
        property => pi_derive_all,
        value    => prop_value);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_node,
        property => pi_import,
        value    => prop_value);

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "menu_t");
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(name, "menu_id");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => menu_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_int4, 2));

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "Menu 2");
    Text_Mgt.Set(name, "menu_title");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => menu_node,
        node_name   => name,
        property    => pi_has_value,
        value       => prop_value.simple_pv);

    Text_Mgt.Set(name, "item_list");
    item_list_node := Data_Definition_Mgt.Create_field(
        record_node => menu_node,
        node_name   => name);

    -- Now create menu items for menu 2:

    -- Create menu item 1 for menu 2:
    --
    Text_Mgt.Set(name, "item_node");
    item_node := Data_Definition_Mgt.Create_node(
        DDef      => ddf,
        node_name => name,
        root      => private_root_node);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_derive_all,
        value    => prop_value);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_import,
        value    => prop_value);

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "menu_item_t");
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_DDef_name,
```

```
        value    => prop_value);

    Text_Mgt.Set(name, "item_id");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_int4, 1));

    Text_Mgt.Set(name, "checked");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_boolean, true));

    Text_Mgt.Set(name, "enabled");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_boolean, true));

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "Menu Item 1");
    Text_Mgt.Set(name, "text");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => prop_value.simple_pv);


    -- Add menu item 1 to menu 2:
    --
    prop_value.simple_pv := (pv_node_reference, item_node);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_list_node,
        property => pi_has_value,
        value    => prop_value);


    -- Create menu item 2 for menu 2:
    --
    Text_Mgt.Set(name, "item_node");
    item_node := Data_Definition_Mgt.Create_node(
        DDef      => ddf,
        node_name => name,
        root      => private_root_node);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_derive_all,
        value    => prop_value);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_import,
        value    => prop_value);

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "menu_item_t");
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(name, "item_id");
    dont_care_node := Data_Definition_Mgt.Create_field(
```

```
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_int4, 2));

    Text_Mgt.Set(name, "checked");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_boolean, true));

    Text_Mgt.Set(name, "enabled");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_boolean, true));

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "Menu Item 2");
    Text_Mgt.Set(name, "text");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => prop_value.simple_pv);


    -- Add menu item 2 to menu 2:
    --
    prop_value.simple_pv := (pv_node_reference, item_node);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_list_node,
        property => pi_has_value,
        value    => prop_value);


    -- Create menu item 3 for menu 2:
    --
    Text_Mgt.Set(name, "item_node");
    item_node := Data_Definition_Mgt.Create_node(
        DDef      => ddf,
        node_name => name,
        root      => private_root_node);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_derive_all,
        value    => prop_value);

    prop_value.simple_pv := (pv_boolean, true);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_import,
        value    => prop_value);

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "menu_item_t");
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_node,
        property => pi_DDef_name,
        value    => prop_value);

    Text_Mgt.Set(name, "item_id");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_int4, 3));
```

```
    Text_Mgt.Set(name, "checked");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_boolean, true));

    Text_Mgt.Set(name, "enabled");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => (pv_boolean, false));

    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "Menu Item 3");
    Text_Mgt.Set(name, "text");
    dont_care_node := Data_Definition_Mgt.Create_field(
        record_node => item_node,
        node_name   => name,
        property    => pi_has_value,
        value       => prop_value.simple_pv);


    -- Add menu item 3 to menu 2:
    --
    prop_value.simple_pv := (pv_node_reference, item_node);
    Data_Definition_Mgt.Add_property_value(
        node_ref => item_list_node,
        property => pi_has_value,
        value    => prop_value);


    -- Add menu 2 to the menu group:
    --
    prop_value.simple_pv := (pv_node_reference, menu_node);
    Data_Definition_Mgt.Add_property_value(
        node_ref => menu_list_node,
        property => pi_has_value,
        value    => prop_value);


    -- Complete and close the menu group:
    --
    prop_value.simple_pv := (pv_type => pv_string);
    Text_Mgt.Set(prop_value.text_value, "/tdo/menu_group_tdo");
    Data_Definition_Mgt.Add_property_value(
        node_ref => group_node,
        property => pi_kind,
        value    => prop_value);


    -- Close the definition (DDef):
    --
    Data_Definition_Mgt.Close(
        DDef => ddf);


    -- Store the DDef:
    --
    Text_Mgt.Set(name, "///pathname/menu_group_DDef");
    Directory_Mgt.Store(name, untyped_ddf);

    -- Request update of stored DDef:
    --
    Passive_Store_Mgt.Request_update(
        obj => untyped_ddf);

end Make_menu_group_DDef_ex;
```

X-A.2.5 `Manage_application_environment_ex` Procedure

```
with CL_Defs,
     Environment_Mgt,
     String_List_Mgt,
     System,
     System_Defs,
     Text_IO,
     Text_Mgt;

procedure Manage_Application_Environment_Ex
  --
  -- Function:
  --   Example program showing use of environment
  --   variables.
  --
  -- History:
  --   06-26-87, William Anton Rohm:  Written.
  --   12-02-87, WAR:                 Revised.
  --
is

  package Int_IO is new Text_IO.Integer_IO(integer);

  -- Variables:
  --
  variable_name:  System_Defs.text(
      CL_Defs.max_name_sz);
  variable_type:  CL_Defs.var_type;
  variable_mode:  CL_Defs.var_mode;

  variable_name_list:  System_Defs.string_list(1000);

  integer_value:  integer;
  ASCII_value:    System_Defs.text(1000);
  answer:         character;

  use CL_Defs;  -- to import "=" for CL_Defs.var_mode
  use System;   -- to import "+" for System.ordinal

begin

  -- Create a new local integer variable named
  -- "new_integer":
  --
  Text_Mgt.Set(
      dest   => variable_name,
      source => "new_integer");

  Environment_Mgt.Set_integer(
      var_name => variable_name,
      value    => 0,
      mode     => CL_Defs.read_write,
      global   => false);


  -- Display all local variable names:
  --
  Environment_Mgt.Get_all_names(
      group_name => System_Defs.null_text,
      list       => variable_name_list,
      global     => false);

  Text_IO.Put_line("List of local variables:");

  for i in 1 .. variable_name_list.count loop

    String_List_Mgt.Get_element(
        from    => variable_name_list,
        el_pos  => i,
        element => variable_name);

    Text_IO.Put_line(variable_name.value);

  end loop;

```

```
-- Read type and mode of given variable:
--    If integer and read-write, add one to variable;
--    otherwise, read and display ASCII
--    representation of value:
--
Text_IO.Put("Enter a variable name:" );

Text_IO.Get(variable_name.value);

variable_type := Environment_Mgt.Get_var_type(
    var_name => variable_name);

variable_mode := Environment_Mgt.Get_var_mode(
    var_name => variable_name);

if variable_type = CL_Defs.integer_type then

  integer_value := Environment_Mgt.Get_integer(
      var_name => variable_name);

  Text_IO.Put("Original value of ");
  Text_IO.Put(variable_name.value);
  Text_IO.Put(" integer variable is:");
  Int_IO.Put(integer_value);
  Text_IO.Put_line(" ");

  if variable_mode = CL_Defs.read_write then
    integer_value := integer_value + 1;

    Environment_Mgt.Set_integer(
        var_name => variable_name,
        value    => integer_value);

    Text_IO.Put("New value of ");
    Text_IO.Put(variable_name.value);
    Text_IO.Put(" integer variable is:");
    Int_IO.Put(integer_value);
    Text_IO.Put_line(" ");

  else
    Text_IO.Put("Mode of ");
    Text_IO.Put(variable_name.value);
    Text_IO.Put_line(" integer variable is 'read-only'.");

  end if;               -- if "read_write"

else                    -- not "integer_type"

  Environment_Mgt.Convert_and_get(
      var_name => variable_name,
      value    => ASCII_value);

  Text_IO.Put("Value of ");
  Text_IO.Put(variable_name.value);
  Text_IO.Put(" variable is:");
  Text_IO.Put_line(ASCII_value.value);

  if variable_mode = CL_Defs.read_write then

      Text_IO.Put("Change value?");

      Text_IO.Get(answer);

      if answer = 'y' or
         answer = 'Y' then

          Text_IO.Put("Enter new value:");
          Text_IO.Get(ASCII_value.value);

          Environment_Mgt.Convert_and_set(
              var_name => variable_name,
              value    => ASCII_value,
              var_type => variable_type);

      end if;           -- if answer = 'y'

```

```
    else
      Text_IO.Put("Mode of ");
      Text_IO.Put(variable_name.value);
      Text_IO.Put_line(" variable is 'read-only'.");

    end if;                         -- if mode = read_write

  end if;                           -- if "integer_type"


  -- Remove new variable:
  --
  Text_Mgt.Set(
      dest   => variable_name,
      source => "new_integer");

  Environment_Mgt.Remove(
      var_name => variable_name,
      quiet    => true,
      global   => false);

end Manage_Application_Environment_Ex;
```

X-A.2.6 `String_list_ex` Procedure

```
with String_List_Mgt,
     System_Defs;

procedure String_list_ex
  --
  -- Function:
  --   Create string list with following entries:
  --     1. "ux_group"
  --     2. "world"
is
  string_list:  System_Defs.string_list(255);
begin

  -- 1) "ux_group"
  String_List_Mgt.Set(string_list,
      System_Defs.text'(8, 8, "ux_group"));

  -- 2) "world"
  String_List_Mgt.Append(string_list,
      System_Defs.text'(5, 5, "world"));

end String_list_ex;
```

X-A.3 Directory Services

X-A.3.1 `Create_directory_cmd_ex` Procedure

```
with Command_Handler,
     Device_Defs,
     Directory_Mgt,
     System_Defs;

procedure Create_directory_cmd_ex
  --
  -- Function:
  --   Creates a named subdirectory in the
  --   caller's current directory.
  --
  -- Command Definition:
  --   The command has the form:
  --     create.directory  :name=<string>
  --
  --   Create the command definition by entering:
  --|    clex -> manage.program  :tagged_source=create.dir.sb
  --
  --*D* set.program  create.directory
  --*D*
  --*D* manage.commands
  --*D*
  --*D*   create.invocation_command
  --*D*     define.argument name  :type = string
  --*D*       set.lexical_class symbolic_name
  --*D*       set.maximum_length 252
  --*D*       set.mandatory
  --*D*       set.description  :text = "
  --*D*         -- Name of directory to be created.
  --*D*       "
  --*D*     end
  --*D*     set.description  :text = "
  --*D*       -- Creates a directory in the
  --*D*       -- current directory.
  --*D*     "
  --*D*   end
  --*D*   exit       -- manage.commands
  --*D* exit       -- manage.program
  --
is

  opened_command:  Device_Defs.opened_device;
    -- Opened invocation command input device.

  dir_name:  System_Defs.text(252);
    -- Name of the directory to be created.

  dir_AD:  Directory_Mgt.directory_AD;
    -- Newly created directory's AD; returned
    -- but not used by "create.directory".
begin

  -- Open invocation command input device:
  --
  opened_command := Command_Handler.
      Open_invocation_command_processing;

  -- Get ":name" parameter:
  --
  Command_Handler.Get_string(
     cmd_odo    => opened_command,
     arg_number => 1,
     arg_value  => dir_name);

  -- Close invocation command input device:
  --
  Command_Handler.Close(opened_command);


  -- Create new named directory:
  --
  dir_AD := Directory_Mgt.Create_directory(
      name => dir_name);

```

```
end Create_directory_cmd_ex;
```

X-A.3.2 Create_name_space_cmd_ex Procedure

```
with CL_Defs,
     Command_Handler,
     Device_Defs,
     Directory_Mgt,
     Environment_Mgt,
     Example_Messages,  -- Example package.
     Incident_Defs,
     Message_Services,
     Name_Space_Mgt,
     Passive_Store_Mgt,
     String_List_Mgt,
     System,
     System_Defs,
     System_Exceptions,
     Transaction_Mgt;

procedure Create_name_space_cmd_ex
  --
  -- Function:
  --   Defines a command to create a name space,
  --   along with the code that executes the command.
  --
  -- Command Definition:
  --   The command has the form:
  --
  --     create.name_space
  --         :name=<string>
  --         :directory_list=<string_list>
  --       [ :force=<boolean>:=false]
  --
  --   Pathnames in the directory list must name
  --   directories.
  --
  --   If "force" is omitted or false then the "name"
  --   pathname must not be in use.  If "force" is
  --   true and the "name" pathname is in use, then
  --   the environment variable "user.confirm" is
  --   consulted.  If "user.confirm" is true (or does
  --   not exist), then the user is queried before
  --   deleting the existing use of the pathname.
  --
  --*C* set.message_file \
  --*C*     :file = /examples/msg/example_messages
  --*C*
  --*C* create.command \
  --*C*     :cmd_def = create.n_s.inv_cmd \
  --*C*     :cmd_name = create.name_space
  --*C*
  --*C*   define.argument name \
  --*C*       :type = string
  --*C*       set.lexical_class symbolic_name
  --*C*       set.maximum_length 252
  --*C*       set.mandatory
  --*C*     end
  --*C*
  --*C*   define.argument directory_list \
  --*C*       :type = string_list
  --*C*       set.lexical_class symbolic_name
  --*C*       set.maximum_length 508
  --*C*     end
  --*C*
  --*C*   define.argument force \
  --*C*       :type = boolean
  --*C*       set.value_default false
  --*C*     end
  --*C* end
  --*C*
  --*C* run "store.command_definitions \\
  --*C*     :exec_unit = create.n_s \\
  --*C*     :invocation_cmd = create.n_s.inv_cmd"
  --*C*
  --*C* run "store.default_message_file \\
  --*C*      create.n_s \\
  --*C*         /examples/msg/example_messages
```

```
  --


is

  opened_cmd:  Device_Defs.opened_device;
    -- Opened command input device.

  name:  System_Defs.text(Incident_Defs.txt_length);
    -- Pathname of new name space.

  directory_list:  System_Defs.string_list(508);
    -- String list containing pathnames of the
    -- directories in the new name space.

  force:  boolean;
    -- Whether the new name space's pathname should
    -- overwrite an existing entry.

  i:  natural;
    -- Index into "directory_list".

  directory_path:  System_Defs.text(Incident_Defs.txt_length);
    -- Text containing each successive pathname from
    -- "directory_list".

  valid:  boolean := true;
    -- True if "directory_list" is valid.  Assigned
    -- false if it is invalid.

  name_space:  Name_Space_Mgt.name_space_AD;
    -- The new name space.

  name_space_untyped:  System.untyped_word;
  FOR name_space_untyped USE AT name_space'address;
    -- The new name space as an untyped word.

  user_confirm_name:  constant System_Defs.text(
      12) := (12, 12, "user.confirm");
    -- Text record of an environment variable's name.

  user_confirm_var_exists:  boolean;
    -- Whether a user variable named
    -- "user.confirm" exists.

  user_confirm_var:  boolean;
    -- Value of "user.confirm" variable, if it exists
    -- ("user_confirm_var_exists" is true).

  overwrite:  boolean;
    -- Whether the created name space can overwrite an
    -- existing entry with the same pathname.


begin

  -- Get command arguments:
  --
  opened_cmd := Command_Handler.
      Open_invocation_command_processing;

  -- Get first argument (name of new name space):
  --
  Command_Handler.Get_string(opened_cmd, 1,
      arg_value => name);

  -- Get second argument (list of directories):
  --
  Command_Handler.Get_string_list(opened_cmd, 2,
      arg_value => directory_list);

  -- Get third argument (force overwrite):
  --
  force := Command_Handler.Get_boolean(opened_cmd, 3);


  Command_Handler.Close(opened_cmd);
```

```
-- Check each pathname in the directory list:
--
i := 1;

loop

  String_List_Mgt.Get_element_by_index(
      from       => directory_list,
      list_index => i,
      element    => directory_path);

  -- Exit after last string:
  --
  EXIT when i = 0;

  -- Check if pathname exists, and is a directory:
  --
  begin
    if not Directory_Mgt.Is_directory(
        Directory_Mgt.Retrieve(directory_path)) then

      valid := false;

      Message_Services.Write_msg(
          Example_Messages.not_directory_code,
          Incident_Defs.message_parameter(
              typ => Incident_Defs.txt,
              len => directory_path.length)'(
                  typ     => Incident_Defs.txt,
                  len     => directory_path.length,
                  txt_val => directory_path));
    end if;

  exception
    when Directory_Mgt.no_access =>

      valid := false;

      Message_Services.Write_msg(
          Example_Messages.no_access_code,
          Incident_Defs.message_parameter(
              typ => Incident_Defs.txt,
              len => directory_path.length)'(
                  typ     => Incident_Defs.txt,
                  len     => directory_path.length,
                  txt_val => directory_path));

  end;

end loop;

if not valid then
  Message_Services.Write_msg(
      Example_Messages.
          create_name_space_aborted_code);
else
  name_space := Name_Space_Mgt.Create_name_space(
      directory_list);

  -- Store new name space as a directory entry:
  --
  loop
    begin

      -- Start a transaction to store new name space:
      --
      Transaction_Mgt.Start_transaction;
      Directory_Mgt.Store(name, name_space_untyped);

      -- Exit if no exception raised:
      --
      EXIT;

    exception

```

```
when System_Exceptions.
    transaction_timestamp_conflict =>

  Transaction_Mgt.Abort_transaction;


when Directory_Mgt.entry_exists =>

  Transaction_Mgt.Abort_transaction;

  if force then

    begin
      user_confirm_var := Environment_Mgt.Get_boolean(
          user_confirm_name);

      user_confirm_var_exists := true;

    exception
        when CL_Defs.non_existent |
             CL_Defs.invalid_type |
             CL_Defs.no_value     =>
            user_confirm_var_exists := false;
    end;

    if user_confirm_var_exists and then
        (not user_confirm_var) then
      -- No confirmation necessary:
      --
      overwrite := true;

    else
      -- Confirm overwrite:
      --
      overwrite :=
          Message_Services.Acknowledge_msg(
              Example_Messages.
                overwrite_query_code,
              Incident_Defs.
                message_parameter(
                typ => Incident_Defs.txt,
                len => name.max_length)'(
                  typ     =>
                      Incident_Defs.txt,
                  len     =>
                      name.max_length,
                  txt_val => name));
    end if;

  else
      -- "force" false:
      --
    overwrite := false;
  end if;

  if overwrite then
    begin
      Directory_Mgt.Delete(name);

    exception
      when Directory_Mgt.no_access =>
        null;
    end;

  else
    Message_Services.Write_msg(
        Example_Messages.not_overwritten_code,
        Incident_Defs.message_parameter(
            typ => Incident_Defs.txt,
            len => name.max_length)'(
                typ     => Incident_Defs.txt,
                len     => name.max_length,
                txt_val => name));

    Message_Services.Write_msg(
        Example_Messages.
            create_name_space_aborted_code);
```

```
     end if;

    when Directory_Mgt.no_access =>

     Transaction_Mgt.Abort_transaction;

     Message_Services.Write_msg(
         Example_Messages.no_access_code,
         Incident_Defs.message_parameter(
             typ => Incident_Defs.txt,
             len => name.max_length)'(
                 typ     => Incident_Defs.txt,
                 len     => name.max_length,
                 txt_val => name));

     Message_Services.Write_msg(
         Example_Messages.
             create_name_space_aborted_code);

    when others =>

     Transaction_Mgt.Abort_transaction;

     RAISE;

   end;

  end loop;

  -- Update passive version:
  --
  Passive_Store_Mgt.Request_update(
      name_space_untyped);

  -- Commit the "store new name space" transaction:
  --
  Transaction_Mgt.Commit_transaction;

  -- Inform user of succesful creation of new name
  -- space:
  --
  Message_Services.Write_msg(
      Example_Messages.name_space_created_code,
      Incident_Defs.message_parameter(
          typ => Incident_Defs.txt,
          len => name.length)'(
              typ     => Incident_Defs.txt,
              len     => name.length,
              txt_val => name));
 end if;       -- if all directories in path are
               -- valid

end Create_name_space_cmd_ex;
```

X-A.3.3 List_current_directory_cmd_ex Procedure

```
with Byte_Stream_AM,
     Command_Handler,
     Device_Defs,
     Directory_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     System,
     System_Defs,
     Unchecked_Conversion;

procedure List_current_directory_cmd_ex
  --
  -- Function:
  --   Lists names of entries in user's current
  --   directory.
  --
  --   Each entry name is written to the user's
  --   standard output, on a separate line.
  --
  -- Command Definition:
  --   The command has the form:
  --     list.current_directory  [:pattern=<string>]
  --
  --*D*  manage.commands
  --*D*     create.invocation_command
  --*D*
  --*D*   define.argument pattern \
  --*D*       :type = string
  --*D*       set.lexical_class symbolic_name
  --*D*       set.maximum_length 252
  --*D*       set.value_default "*"
  --*D*     end
  --*D* end
  --*D*
  --
  --
is

  -- Generic function:
  --
  function Directory_AD_from_untyped_word is
      new Unchecked_conversion(
          source => System.untyped_word,
          target => Directory_Mgt.directory_AD);


  -- Variables:
  --
  odo:  Device_Defs.opened_device :=
      Command_Handler.
          Open_invocation_command_processing;
    -- Opened invocation command input device.

  pattern:  System_Defs.text(252) := (252, 252, (others => ' '));
    -- Optional ":pattern" used to select entries
    -- matching the pattern, such as "abc?" or
    -- "m*device".  Default is "!.*", meaning all
    -- entries NOT beginning with a "." (period).

  opened_dir:  Device_Defs.opened_device;
    -- Opened device for reading stream of names
    -- from user's current directory.

  standard_output:  Device_Defs.opened_device :=
      Device_Defs.opened_device(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.standard_output));
    -- User's standard output.

  name_buffer:  array(1 .. 250) of character;
    -- Each entry name is read into this buffer
    -- and then written from it.

  length:  System.ordinal;
```

```
    -- Length in bytes (characters) of last
    -- entry name read.
use System;      -- for " 'size/8 " arithmetic

begin

  -- Get ":pattern", if any:

  Command_Handler.Get_string(
     cmd_odo    => odo,
     arg_number => 1,
     arg_value  => pattern);

  -- Close invocation command input device:
  --
  Command_Handler.Close(odo);

  -- Open directory for reading, filtered by
  -- ":pattern":
  --
  opened_dir := Directory_Mgt.Open_directory(
      dir     => Directory_AD_from_untyped_word(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.current_dir)),
      pattern => pattern);


  -- Get and write each entry name:
  --
  loop

    length := Byte_Stream_AM.Ops.Read(
        opened_dev => opened_dir,
        buffer_VA  => name_buffer'address,
        length     => name_buffer'size/8);

    Byte_Stream_AM.Ops.Write(
        opened_dev => standard_output,
        buffer_VA  => name_buffer'address,
        length     => length);

  end loop;

exception

  when Device_Defs.end_of_file =>

    Byte_Stream_AM.Ops.Close(opened_dir);

    RETURN;

end List_current_directory_cmd_ex;
```

X-A.3.4 Make_object_public_ex Procedure

```
with Authority_List_Mgt,
     Directory_Mgt,
     Identification_Mgt,
     Passive_Store_Mgt,
     System,
     System_Defs,
     Transaction_Mgt,
     User_Mgt;

procedure Make_object_public_ex(
    obj:              System.untyped_word;
      -- Object to be made public.
    aut_list_path:  System_Defs.text)
      -- Pathname under which to store the new
      -- authority list.
  --
  -- Function:
  --   Makes an object "public" by giving it an
  --   authority list that grants all type rights
  --   to the "world" ID.
  --
  -- Logic:
  --   1. Get an AD to the world ID.
  --   2. Define a protection set that grants all
  --      type rights to the world ID.
  --   3. Create an authority list with that
  --      protection set.
  --   4. Enclose steps (5) and (6) in a transaction.
  --   5. Store the authority list under the pathname
  --      given as the "aut_list_path" parameter.
  --   6. Passivate the authority list, so that it
  --      will endure in passive store along with
  --      the object that it protects.
  --   7. Assign the authority list as the object's
  --      authority list.
  --
  -- Exceptions:
  --   Authority_List_Mgt.set_authority_refused -
  --     The object's master AD was stored with
  --     no authority list protecting the object,
  --     and an authority list cannot now be assigned.
is
  -- Get the world ID AD
  world_name:  constant System_Defs.text(9) :=
      (9, 9, "/id/world");
  world_untyped:  constant System.untyped_word :=
      Directory_Mgt.Retrieve(world_name);
  world_id:  Identification_Mgt.ID_AD;
  FOR world_id USE AT world_untyped'address;

  -- Define the protection set
  entries:   constant User_Mgt.protection_set(1) := (
      size  => 1, length => 1,
      entries => (1 => (rights => (true, true, true),
                        id     => world_id)));

  -- Create the authority list
  aut_list:  constant
      Authority_List_Mgt.authority_list_AD :=
      Authority_List_Mgt.Create_authority(entries);
  aut_untyped:  System.untyped_word;
  FOR aut_untyped USE AT aut_list'address;

begin
  Transaction_Mgt.Start_transaction;
  begin
    Directory_Mgt.Store(aut_list_path, aut_untyped);
    Passive_Store_Mgt.Request_update(aut_untyped);
    Transaction_Mgt.Commit_transaction;
  exception
    when others =>
      Transaction_Mgt.Abort_transaction;
      RAISE;

```

```
  end;
  Authority_List_Mgt.Set_object_authority(
      obj, aut_list);
end Make_object_public_ex;
```

X-A.3.5 Show_current_directory_cmd_ex Procedure

```
with Byte_Stream_AM,
     Device_Defs,
     Directory_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     System,
     System_Defs,
     Text_Mgt;

procedure Show_current_directory_cmd_ex
  --
  -- Function:
  --   Gets and displays the pathname of the
  --   current directory.
  --
  -- Command Definition:
  --   The command has the form:
  --     show.current_directory
  --
  --*C* create.command \
  --*C*     :cmd_def = show.cur_dir.inv_cmd \
  --*C*     :cmd_name = show.current_directory
  --*C* end
  --*C*
  --*C* run "store.command_definitions \\
  --*C*     :exec_unit = show.cur_dir \\
  --*C*     :invocation_cmd = show.cur_dir.inv_cmd"
  --
is

  standard_output:  Device_Defs.opened_device :=
      Device_Defs.opened_device(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.standard_output));
    -- User's standard output.

  current_dir:   Directory_Mgt.directory_AD :=
      Directory_Mgt.directory_AD(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.current_dir));
    -- Current directory's AD.

  current_dir_untyped: System.untyped_word;
     FOR current_dir_untyped USE AT
         current_dir'address;
    -- Current directory's AD as an untyped word.

  dir_name:  System_Defs.text(252);
    -- Current directory's name.

begin

  -- Get current directory's pathname:
  --
  Directory_Mgt.Get_name(
      obj  => current_dir_untyped,
      name => dir_name);

  -- Add a line-feed to pathname for displaying:
  --
  Text_Mgt.Append(
      dest   => dir_name,
      source => Standard.ASCII.LF);

  -- Display pathname:
  --
  Byte_Stream_AM.Ops.Write(
     opened_dev => standard_output,
     buffer_VA  => dir_name.value'address,
     length     => System.ordinal(
         dir_name.length));

end Show_current_directory_cmd_ex;
```

X-A.4 I/O Services

X-A.4.1 DBMS_Support_Ex Package Specification

```
with Device_Defs,
     System,
     System_Defs;

package DBMS_Support_EX is
  --
  -- Function:
  --   Shows how to use the record processing and
  --   DBMS support operations in applications.
  --
  -- History:
  --   08-15-87, Paul Schwabe: initial version.
  --   12-01-87, Paul Schwabe: reorganized.
  --
  pragma external;

  procedure Selection(
      opened_file:   Device_Defs.opened_device;
      read_procedure:  System.subprogram_type);
        -- An opened device, opened for input on an
        -- employee file.
    --
    -- Function:
    --   Do a Record_AM.Keyed_Ops.Set_key_range using
    --   the Dept index. Do a
    --   Record_Processing_Support.Set_oriented_read.
    --   Returns a set of records for the range of
    --   departments indicated.


  procedure Projection(
      opened_file:        Device_Defs.opened_device;
      projection_DDef_name: System_Defs.text);
        -- An opened device, opened for input on an
        -- employee file.

    -- Function:
    --   Grabs only certain fields for each record
    --   that is read from the employee file.  Set
    --   the filter up using the following call:


  procedure Sort_records(
      inventory_file:  Device_Defs.opened_device;
      inventory_DDef_name: System_Defs.text);
        --   An opened device, opened for input on an
        --   inventory file. Uses
    --
    -- Function:
    --   Sort_Merge_Interface.Sort to sort records
    --   from an inventory file (writes to standard
    --   out).


  procedure Merge_and_sort_records(
      inventory_file:  Device_Defs.opened_device;
      employee_file:   Device_Defs.opened_device;
      sort_DDef_name: System_Defs.text);
        -- Two opened devices, opened for input on an
        -- inventory file and employee file.
    --
    -- Function:
    --   Uses Sort_Merge_Interface.Sort_merge to merge
    --   and sort records from two (the inventory and
    --   the employee) files (writes to standard out).



end DBMS_Support_EX;
```

X-A.4.2 DBMS_Support_Ex Package Body

```
with Employee_Filing_Ex,
     Data_Definition_Mgt,
     Device_Defs,
     Process_Globals_Support_Ex,
     Record_AM,
     Record_Processing_Support,
     Sort_Merge_Interface,
     Trusted_Record_Processing_Support,
     System,
     System_Defs,
     Unchecked_conversion;

use System;

package body DBMS_Support_Ex is
  --
  -- Logic:
  --  Shows how to do record processing
  --  support operations.
  --


  procedure Selection(
     opened_file:     Device_Defs.opened_device;
     read_procedure:  System.subprogram_type)
       -- An opened device, opened for input on an
       -- employee file.
    -- Logic:
    --   Do a Record_AM.Keyed_Ops.Set_key_range using
    --   the Dept index. Do a
    --   Record_Processing_Support.Set_oriented_read.
    --   Returns a set of records for the range of
    --   departments indicated.
  is
    start_key_value:  constant
       Employee_Filing_Ex.dept_key_buffer := (
         dept       => 100);
           -- Lowest dept for ascending key field.

    start_key_descr: constant
        Record_AM.key_value_descr := (
            start_key_value'address,
            start_key_value'size / 8);

    stop_key_value: constant
        Employee_Filing_Ex.dept_key_buffer := (
            dept => 305);
          -- Highest dept value
          -- for ascending key field.

    stop_key_descr:  constant
        Record_AM.key_value_descr := (
            stop_key_value'address,
            stop_key_value'size / 8);

  begin
    Trusted_Record_Processing_Support.Associate_read_procedure(
        opened_dev          => opened_file,
        user_info           => System.null_address,
        read_procedure      => read_procedure);


    Record_AM.Keyed_Ops.Set_key_range(
        opened_dev    => opened_file,
        index         =>
            Employee_Filing_Ex.dept_index_name,
        select_range => (
            start_comparison => Record_AM.inclusive,
            start_value      => start_key_descr,
            stop_comparison  => Record_AM.inclusive,
            stop_value       => stop_key_descr));

    Record_Processing_Support.Set_oriented_read(
        opened_dev     => opened_file,
```

```
        modifier       => Record_AM.next,
        output_device  => Process_Globals_Support_Ex.
            Get_standard_output,
          -- Normally defaulted.
        alt_output     => System.null_word,
        no_record_lock => false,
        lock           => Record_AM.read_lock,
        unlock         => Record_AM.no_unlock,
        timeout        => Record_AM.wait_forever);
      -- DO ANY NEEDED PROCESSING HERE.

  exception
    when Device_Defs.end_of_file =>
      null;

  end Selection;


  procedure Projection(
      opened_file: Device_Defs.opened_device;
      projection_DDef_name: System_Defs.text)
       -- An opened device, opened for input on an
       -- employee file.
    -- Logic:
    --   Grabs only certain fields for each record
    --   that is read from the employee file.
    --
  is
    projection_DDef_ref:    Data_Definition_Mgt.
                                node_reference;

    buffer:  string(1 .. integer(Employee_Filing_Ex.max_rec_size));
      -- Buffer is large enough to hold any employee
      -- record.

    current_record_addr:  constant
        System.address := buffer'address;
    current_record_VA:  constant
        Employee_Filing_Ex.employee_record_VA :=
           Employee_Filing_Ex.
               Employee_record_VA_from_VA(
                  current_record_addr);

    bytes_read:  System.ordinal;
      -- Number of bytes in current record.

  begin
    --
    -- Open projection data definition.
    --

    projection_DDef_ref :=
        Record_AM.Ops.Get_DDef(
            opened_dev => Record_AM.Open_by_name(
                name         => projection_DDef_name,
                input_output => Device_Defs.input,
                allow        => Device_Defs.readers,
                block        => true));

    -- Filters out all fields except those specified
    -- in the DDef.
    Record_Processing_Support.
        Associate_primary_data_projection(
            opened_dev       => opened_file,
            record_ID_output => false,
            primary_fields   => projection_DDef_ref);


    loop
      -- Only reads the fields specified in
      -- the DDef.
      bytes_read := Record_AM.Ops.Read(
          opened_dev => opened_file,
          modifier   => Record_AM.next,
            -- Normally defaulted.
          buffer_VA  => current_record_addr,
          length     => System.ordinal(
```

```
                   Employee_Filing_Ex.max_rec_size));

        -- DO ANY NEEDED PROCESSING HERE.

      end loop;
    exception
      when Device_Defs.end_of_file =>
        null;

    end Projection;



    procedure Sort_records(
        inventory_file:      Device_Defs.opened_device;
        inventory_DDef_name: System_Defs.text)
          -- An opened device, opened for input on an
          -- inventory file.
      -- Logic:
      --   Uses Sort_Merge_Interface.Sort to sort
      --   records from an inventory file (writes to
      --   standard out).
    is
      opened_inventory_ddef: Device_Defs.opened_device;
      inventory_ddef_ref:    Data_Definition_Mgt.
                                 node_reference;
    begin
      --
      -- Open inventory definition.
      --
      opened_inventory_DDef :=
         Record_AM.Open_by_name(
             name         => inventory_DDef_name,
             input_output => Device_Defs.input,
             allow        => Device_Defs.readers,
             block        => true);

      inventory_DDef_ref :=
         Record_AM.Ops.Get_DDef(
             opened_dev => opened_inventory_DDef);

      Sort_Merge_Interface.Sort(
          input_device  => inventory_file,
          DDef          => inventory_DDef_ref,
          output_device => Process_Globals_Support_Ex.
              Get_standard_output,
          stable_sort   => true,
          tuning_opts   => Sort_Merge_Interface.
              no_tuning);
      --
      -- Close inventory file.
      --
      Record_AM.Ops.Close(
          opened_dev => opened_inventory_DDef);

    end Sort_records;


    procedure Merge_and_sort_records(
        inventory_file: Device_Defs.opened_device;
        employee_file:  Device_Defs.opened_device;
        sort_DDef_name: System_Defs.text)
         -- Two opened devices, opened for input on an
         -- inventory file and employee file. Uses
      -- Logic:
      --   Sort_Merge_Interface.Sort_merge to merge
      --   and sort records from two (the inventory
      --   and the employee) files (writes to
      --   standard out).
    is
      opened_sort_DDef: Device_Defs.opened_device;
      sort_DDef_ref:    Data_Definition_Mgt.
                            node_reference;
      sort_input_array: Sort_Merge_Interface.
          sort_merge_input_array(1 .. 2) :=
              (1 => (input_device => inventory_file,
                     presorted    => false,
```

```
                   sorted_by_index => false),
             2 => (input_device => employee_file,
                   presorted    => false,
                   sorted_by_index => false));
  begin
    --
    -- Open sort data definition.
    --
    opened_sort_DDef :=
       Record_AM.Open_by_name(
           name          => sort_DDef_name,
       input_output => Device_Defs.input,
           allow         => Device_Defs.readers,
           block         => true);

    sort_DDef_ref :=
       Record_AM.Ops.Get_DDef(
           opened_dev => opened_sort_DDef);

    -- Perform the sort-merge.
    Sort_Merge_Interface.Sort_merge(
        input_devices => sort_input_array,
        DDef          => sort_DDef_ref,
        output_device => Process_Globals_Support_EX.
            Get_standard_output,
        stable_sort   => true,
        tuning_opts   => Sort_Merge_Interface.
            no_tuning);

    --
    -- Close inventory file.
    --
    Record_AM.Ops.Close(
        opened_dev => opened_sort_DDef);

  end Merge_and_sort_records;


end DBMS_Support_EX;
```

X-A.4.3 `Employee_Filing_Ex` Package Specification

```
with Data_Definition_Mgt,
     File_Defs,
     System,
     System_Defs,
     Unchecked_conversion;

use System;

package Employee_Filing_Ex is

  -- Function:
  --   Defines an employee file structure.
  --
  --
  --   Contains declarations for employee records and
  --   indexes.   Contains  subprograms for creating
  --   needed DDefs and for creating an employee file
  --   with indexes.
  --
  --   The "employee_record" type  defines the record
  --   format.
  --
  --   An employee file has two indexes:
  --
  --   "Dept index" - A b-tree index sorted by salary
  --   ascending department.  Allows  duplicates.
  --
  --   "Dept-salary" index  -  A  b-tree  index
  --   sorted  by  ascending  department and  descending
  --   salary.  Allows duplicates.
  --
  pragma external;

  --
  -- CONSTANTS
  --

  max_text_length:    constant :=  25;
    --  The maximum length for a person's
    --  name.
  max_job_desc_length:  constant :=  200;
    -- The maximum  length  of a  job description
    -- string.

  --
  -- FIELD SUBTYPES OR TYPES

  subtype department_number is
      System.ordinal range 0 .. 1000;
    -- A work group within the company.

  subtype person_name is
     System_Defs.text(max_text_length);

    -- Format is:  last-name, first-name middle-name
    -- [suffix ] This format is used so that records
    -- can be ordered alphabetically on last name then
    -- first name.

  subtype job_description_length is
    integer range 0 .. max_job_desc_length;
      -- String length allowed for a job
      -- description.

  subtype monthly_salary is float;
    -- The monthly salary for an employee.

  --
  -- RECORD DECLARATIONS

  type employee_record(
     length:  job_description_length) is
    record
      dept:        department_number;
```

```
      name:         person_name;
      job_descr:    string(1 .. length);
      salary:       monthly_salary;
    end record;

  -- This specific representation assures the
  -- record is correctly represented for the
  -- DDef.   The fields must be word aligned.

  FOR employee_record USE
    record
      dept      at  0 range  0 .. 31;
      name      at  4 range  0 .. 231;
      salary    at 36 range  0 .. 63;
    end record;


  max_rec_size: constant System.ordinal :=  241;
    -- Maximum number of bytes in the employee record.
    -- Used to determine the buffer size when
    -- reading an employee record.

  type employee_record_VA is access employee_record;
  pragma access_kind(employee_record_VA, virtual);
    -- Type contains virtual pointers to employee
    -- records.

  employee_DDef:  Data_Definition_Mgt.node_reference;
    -- Data definition for the employee record.

  --
  -- DECLARATIONS FOR INDEXES
  -- A simple index declaration.
  dept_index_DDef:  Data_Definition_Mgt.
                         node_reference;

  dept_index_name:  constant
     File_Defs.index_name :=
        (max_length => File_Defs.index_name_length,
         length => 14,
         value  => "Dept_Index_DDef                  ");

  type dept_key_buffer is
    record
      dept:         department_number;
    end record;

   -- A composite index declaration.
   dept_salary_index_DDef:
     Data_Definition_Mgt.node_reference;

  dept_salary_index_name:  constant
     File_Defs.index_name :=
        (max_length => File_Defs.index_name_length,
         length     => 21,
         value      => "Dept_Salary_Index_DDef           ");

  type dept_salary_key_buffer is
    record
      dept:         department_number;
      salary:       monthly_salary;
    end record;
  -- This specific representation assures the
  -- buffer is correctly represented for the
  -- DDef.  There is no padding between fields.
  FOR dept_salary_key_buffer USE
    record
      dept                at  0 range  0 .. 31;
      salary              at  4 range  0 .. 63;
    end record;

  --
  -- CALLS
  --


  function Employee_record_VA_from_VA is new
```

```
  Unchecked_conversion(
      source => System.address,
      target => employee_record_VA);


function VA_from_employee_record_VA is new
  Unchecked_conversion(
      source => employee_record_VA,
      target => System.address);



procedure Create_employee_DDef;
  --
  -- Function:
  --   Creates DDefs for the employee record and all
  --   indexes.
  --
  --   The DDefs are in a single DDef object, which
  --   is passivated with the specified pathname.
  --
  --   "Create_employee_DDefs" assigns all the
  --   "ddef" variables in this package.
  --
  -- Notes:
  --   "Create_employee_DDefs" is normally called
  --   only once in the lifetime of a system.
  --
  --   The same DDefs can be used by multiple
  --   employee files.


procedure Create_dept_DDef;
  --
  -- Function:
  --   Sets up an index key DDef for an employee
  --   file by deriving fields from an existing
  --   record DDef.
  --
  --   The index key DDef requires the properties
  --   indicated by the following pseudo-DDef
  --   language:
  --
  --   define "Dept"
  --     record  Import from (
  --                 "Employee_Data",
  --                 "Employee_DDef"),
  --             Derive_all is false;
  --       Maps to "Dept",
  --   This simple index key is set up by mapping
  --   DDef nodes from "Employee_DDef" to a new
  --   record DDef called "Index_2_DDef"
  --   that consists of one field:
  --     * "Dept" in ascending order.

procedure Create_dept_salary_DDef;
  --
  -- Function:
  --   Sets up an index key DDef for an employee
  --   file by deriving fields from an existing
  --   record DDef.
  --
  --   The index key DDef requires the properties
  --   indicated by the following pseudo-DDef
  --   language:
  --
  --   define "Dept-Salary"
  --     record  Import from (
  --                 "Employee_Data",
  --                 "Employee_DDef"),
  --             Derive_all is false;
  --       Maps to "Dept",
  --       Maps to "Salary",
  --             descending is true;
  --
  --   This composite index key is set up by mapping
  --   DDef nodes from "Employee_DDef" to a new
```

```
    --   record DDef called "Dept-Salary"
    --   that consists of two fields:
    --     * "Dept" in ascending order.
    --     * "Salary" in descending order.


  procedure Create_file_and_indexes(
      file_name:        System_Defs.text;
      org_index_name:  System_Defs.text);
        -- New file's pathname.
    --
    -- Function:
    --   Creates an employee file with all needed
    --   indexes.  The employee file is a clustered
    --   organization.
    --
    --   The new file is initially empty.
    --
    --   "Create_employee_DDefs" must have been called
    --   *before* any call to "Create_employee_file".
    --
    -- Note:
    --   The index is built after the file is created.
    --
    --   The file uses DDefs defined in the
    --   Employee_Filing_Ex package.


end Employee_Filing_Ex;
```

X-A.4.4 Employee_Filing_Ex Package Body

```
with Data_Definition_Mgt,
     Directory_Mgt,
     File_Admin,
     File_Defs,
     Passive_Store_Mgt,
     System,
     System_Defs,
     Text_Mgt;

package body Employee_Filing_Ex is

  max_employee_count:  System.ordinal := 1_000;
    -- A new employee file is limited to this many
    -- employees.


  procedure Store_DDef(
      DDef:     Data_Definition_Mgt.DDef_AD;
      name:     System_Defs.text)
  is
    -- Logic:
    --   Stores a DDef and updates its passive
    --   version.
    --
    untyped_DDef:    untyped_word;
      FOR untyped_DDef USE AT DDef'address;
    --
  begin
    begin
      Directory_Mgt.Delete(name);
    exception
      when Directory_Mgt.no_access =>
        null;

      when others =>
        RAISE;
    end;
    Directory_Mgt.Store(name, untyped_DDef);

    Passive_Store_Mgt.Request_update(untyped_DDef);

  end Store_DDef;

  procedure Create_employee_DDef
        -- New DDef object's pathname.
    --
    -- Logic:
    --   Sets up a self-contained record DDef.  This
    --   DDef requires the properties indicated by
    --   the following pseudo-DDef language:
    --
    --   define Employee_Data
    --     record
    --       Dept:      Type is ord_2,
    --                  lower_bound is 100,
    --                  upper_bound is 999;
    --       Name:      Type is string,
    --                  (System_Defs.text)
    --                  Header_for_max_length is true,
    --                  Varying is true,
    --                  length is 25;
    --       Job_Desc:  Type is string,
    --                  length is 200;
    --       Salary:    Type is real4,
    --                  default_value is 0;
    --     end record;
    --
    --   This structure is equivalent to the following
    --   Ada record declaration:
    --
    --   subtype Job_Desc_length is
    --     integer range 0.. 200;
    --
    --   Employee_Data(
```

```
  --       length: Job_Desc_length) is
  --     record
  --       dept:     short_ordinal range 100 .. 999;
  --       name:     System_Defs.text(25);
  --       job_Desc: string(1 .. length);
  --       salary:   float;
  --     end record;
  --
  --   "Data_Definition_Mgt" assigns layout
  --   properties to the record that correspond to
  --   the following Ada rep spec (note that the
  --   holes in the record allow fields to be placed
  --   on natural boundaries):
  --
  --   for Employee_Data use
  --     record
  --       dept      at  0 range  0 .. 15;
  --       name      at  4 range  0 ..
  --           8*(max_text_length+4)-1;
  --       length    at 40 range  0 .. 15;
  --       job_desc  at 42 range  0 ..
  --           8*(job_desc_length)-1;
  --       salary    at 36 range  0 .. 31;
  --     end record;
is
  dd:         Data_Definition_Mgt.DDef_AD;
  name:       System_Defs.text(40);
  rec_node:   Data_Definition_Mgt.node_reference;
  field_node: Data_Definition_Mgt.node_reference;
  pv: Data_Definition_Mgt.property_value(100);
begin

  dd := Data_Definition_Mgt.Create_DDef;
    -- Create a new DDef object.


  Text_Mgt.Set(name,"Employee_Data");
  rec_node := Data_Definition_Mgt.Create_node(
    -- Create a DDef node for the record layout.
      dd,
        -- AD to a DDef object
      Data_Definition_Mgt.mt_record,
        -- Record metatype and property value for
        -- the "node_name" property ID.
      name,
      Data_Definition_Mgt.public_root_node);
        -- Property value for the "root_value"
        -- property ID.


  Text_Mgt.Set (name,"Dept");

  -- Create a simple metatype node with
  -- "root_value" set to "non_root_node" for the
  -- "Dept" field.
  field_node := Data_Definition_Mgt.
      Create_simple_field(
          rec_node,
            --  DDef object open for definition.
          Data_Definition_Mgt.t_ord2,
            -- Property value for "pi_type" property
            -- ID (short ordinal of type "type_t").
          name);
            -- Property value for the "node_name"
            -- property ID.

  pv.simple_pv := (
      pv_type => Data_Definition_Mgt.pv_int4,
      int4_value => 100);
        -- Set "pi_lower_bounds" (type integer) to
        -- 100.

  -- Add "pi_lower_bounds" and its value to the
  -- "Dept" node.
  Data_Definition_Mgt.Add_property_value(
      field_node,
      Data_Definition_Mgt.pi_lower_bounds,
```

```
           pv);

       pv.simple_pv.int4_value := 999;
         -- Set "upper_bounds" property value.

       Data_Definition_Mgt.Add_property_value(
         -- Add "pi_upper_bounds" and its value to the
         -- node.
           field_node,
           Data_Definition_Mgt.pi_upper_bounds,
           pv);


       Text_Mgt.Set (name,"Name");

       -- Create a simple metatype node with
       -- "root_value" set to "non_root_node" for the
       -- "Name" field.
       field_node := Data_Definition_Mgt.
           Create_simple_field_with_prop(
               rec_node,
                 -- DDef object that is open for
                 -- definition.
               Data_Definition_Mgt.t_string,
                 -- Value for "pi_type" (uses byte-string
                 -- for "type_t").
               name,
                 --  Value for "node_name".
               Data_Definition_Mgt.
                   pi_header_for_max_length,
               (Data_Definition_Mgt.pv_boolean,true));
                 -- True if string is represented in
                 -- SIL 'text' type.

       pv.simple_pv := (
           pv_type => Data_Definition_Mgt.pv_int4,
           int4_value => 25);
             -- Property value (type integer) is set to
             -- 25.

       Data_Definition_Mgt.Add_property_value(
               field_node,
                  -- Node within an open DDef object.
               Data_Definition_Mgt.pi_length,
               pv);
         -- Sets "pi_length" (maximum length of string in
         -- bytes). Because  "pi_header_for_max_length"
         -- requires "pi_varying" to be false, "name" is
         -- a fixed-size field.


       Text_Mgt.Set (name,"Job_Desc");

       -- Create a simple metatype node with
       -- "root_value" set to "non_root_node".
       field_node := Data_Definition_Mgt.
           Create_simple_field_with_prop(
                rec_node,
                   --   DDef object that is open for
                   --   definition.
                Data_Definition_Mgt.t_string,
                   -- Value for "pi_type" (uses
                   -- byte-string for "type_t").
                name,
                   -- Value for "pi_node_name".
                Data_Definition_Mgt.pi_varying,
                    (Data_Definition_Mgt.pv_boolean,
                     true));
                  -- Varying-length string.

       pv.simple_pv := (
           pv_type => Data_Definition_Mgt.pv_int4,
             -- Sets property value for "pi_length"
             -- (maximum length of string in bytes) to
             -- 200.
           int4_value => 200);
```

```
  Data_Definition_Mgt.Add_property_value(
    -- Adds "pi_length" and its value.
          field_node,
            -- Node within an open DDef object.
          Data_Definition_Mgt.pi_length,
          pv);


  Text_Mgt.Set (name,"Salary");
  field_node := Data_Definition_Mgt.
      Create_simple_field_with_prop(
        -- Create a simple metatype node with
        -- "root_value" set to "non_root_node"
        -- (defaults to 0).
          rec_node,
            -- DDef object that is open for
            -- definition.
          Data_Definition_Mgt.t_real8,
            -- Value for "pi_type"
            -- (uses real for "type_t").
          name,
            -- Value for "pi_node_name".
          Data_Definition_Mgt.pi_default_value,
          (Data_Definition_Mgt.pv_real8,0.0));

  Data_Definition_Mgt.Close(dd);
    -- Close and bind DDef object.

  -- Save created DDef as "Employee_DDef".
  Text_Mgt.Set(name,"Employee_DDef");
  Store_DDef(DDef => dd, name => name);

end Create_employee_DDef;


procedure Create_dept_DDef
  --
  -- Logic:
  --   Sets up an index key DDef for an employee
  --   file by deriving fields from an existing
  --   record DDef.
  --
is
  dd:         Data_Definition_Mgt.DDef_AD;
  name:  System_Defs.text(40);
  rec_node:   Data_Definition_Mgt.node_reference;
  field_node: Data_Definition_Mgt.node_reference;
  pv:         Data_Definition_Mgt.
                  property_value(100);
begin
  -- Create AD to a DDef object
  dd := Data_Definition_Mgt.Create_DDef;
  -- Create node for Index_2_DDef record
  Text_Mgt.Set(name,"Index_2_DDef");
  rec_node := Data_Definition_Mgt.Create_node(
      dd,
        --  AD to a DDef object
      Data_Definition_Mgt.mt_record,
          --  meta_type of 'record'
      name,
        -- value for the node_name
        -- property
      Data_Definition_Mgt.private_root_node);
        -- can be referenced from
        -- other DDef objects

  -- Set DDef_name property
  pv.simple_pv := (
    pv_type => Data_Definition_Mgt.pv_string);

  Text_Mgt.Set (pv.text_value, "Employee_Data");

  Data_Definition_Mgt.Add_property_value(
      rec_node,
            -- node within an open DDef
      Data_Definition_Mgt.pi_DDef_name,
            -- requested property
```

```
          pv);
            -- value to be assigned
      Text_Mgt.Set (pv.text_value, "Employee_DDef");
      Data_Definition_Mgt.Add_property_value(
              rec_node,
                -- node within an open DDef
              Data_Definition_Mgt.pi_DDef_name,
                -- requested property
              pv);
                -- value to be assigned

      -- Set derive_all property; false: all fields not
      -- referred to.
      pv.simple_pv := (
          pv_type => Data_Definition_Mgt.pv_boolean,
            -- property value has type boolean
          boolean_value => false);
      Data_Definition_Mgt.Add_property_value(
          rec_node,
             -- node within an open DDef
          Data_Definition_Mgt.pi_derive_all,
             -- requested property
          pv);
             -- value to be assigned

      -- Create node for key field "Dept"
      field_node := Data_Definition_Mgt.
                        Create_field(rec_node);
        -- first key.

      -- Set maps_to property
      pv.simple_pv := (
          pv_type => Data_Definition_Mgt.pv_string);
      Text_Mgt.Set (pv.text_value, "Dept_DDef");
      Data_Definition_Mgt.Add_property_value(
          field_node,
            -- node within an open DDef
          Data_Definition_Mgt.pi_maps_to,
            -- requested property
          pv);
            -- value to be assigned
            -- Descending defaults to false;
            -- it needn't be set.

      -- close and bind DDef
      Data_Definition_Mgt.Close(dd);

      -- Save created DDef under the symbolic name
      -- "Index_2_DDef"
      Text_Mgt.Set(name,"Dept_Index_DDef");
      Store_DDef(DDef => dd, name => name);


    end Create_dept_DDef;



    procedure Create_dept_salary_DDef
      --
      -- Logic:
      --   Sets up an index key DDef for an employee
      --   file by deriving fields from an existing
      --   record DDef.
      --
    is
      dd:         Data_Definition_Mgt.DDef_AD;
      name:  System_Defs.text(40);
          -- New DDef object's pathname.
      rec_node:   Data_Definition_Mgt.node_reference;
      field_node: Data_Definition_Mgt.node_reference;
      pv:         Data_Definition_Mgt.property_value(100);
    begin
      -- Create AD to a DDef object
      dd := Data_Definition_Mgt.Create_DDef;

      -- Create node for Employee_DDef record
      Text_Mgt.Set(name,"Employee_DDef");
```

```
rec_node := Data_Definition_Mgt.Create_node(
    dd,
      --  AD to a DDef object
    Data_Definition_Mgt.mt_record,
      --  meta_type = record
    name,
      -- Value for the node_name property
    Data_Definition_Mgt.private_root_node);
      -- Can be referenced from other DDef objects.

-- Set DDef_name property
pv.simple_pv := (
    pv_type => Data_Definition_Mgt.pv_string);
Text_Mgt.Set (pv.text_value, "Employee_Data");
Data_Definition_Mgt.Add_property_value(
    rec_node,
      -- Node within an open DDef.
    Data_Definition_Mgt.pi_DDef_name,
      -- Requested property.
    pv);
      -- Value to be assigned.
Text_Mgt.Set (pv.text_value, "Employee_DDef");
Data_Definition_Mgt.Add_property_value(
    rec_node,
      -- Node within an open DDef.
    Data_Definition_Mgt.pi_DDef_name,
      -- Requested property.
    pv);
      -- Value to be assigned.

-- Set derive_all property; false: all fields not
-- referred to.
pv.simple_pv := (
  pv_type => Data_Definition_Mgt.pv_boolean,
    -- property value has type boolean
  boolean_value => false);
Data_Definition_Mgt.Add_property_value(
    rec_node,
       -- node within an open DDef
    Data_Definition_Mgt.pi_derive_all,
       -- requested property
    pv);
       -- value to be assigned

-- Create node for key field "Dept"
field_node := Data_Definition_Mgt.
                  Create_field(rec_node);
  -- first key.

-- Set maps_to property
pv.simple_pv := (
    pv_type => Data_Definition_Mgt.pv_string);
Text_Mgt.Set (pv.text_value, "Dept");
Data_Definition_Mgt.Add_property_value(
    field_node,
      -- node within an open DDef
    Data_Definition_Mgt.pi_maps_to,
      -- requested property
    pv);
      -- value to be assigned
      -- Descending defaults to false;
      -- it needn't be set.

-- Create node for key field "Salary"
field_node := Data_Definition_Mgt.Create_field(
                  rec_node);

-- Set maps_to property
pv.simple_pv := (
    pv_type => Data_Definition_Mgt.pv_string);
Text_Mgt.Set (pv.text_value, "Salary");
Data_Definition_Mgt.Add_property_value(
    field_node,
        -- node within an open DDef
    Data_Definition_Mgt.pi_maps_to,
        -- requested property
    pv);
```

```
            -- value to be assigned

  -- Set descending property; true: order is
  -- descending
  pv.simple_pv := (
      pv_type => Data_Definition_Mgt.pv_boolean,
        -- property value has type boolean
      boolean_value => true);
  Data_Definition_Mgt.Add_property_value(
      field_node,
        -- node within an open DDef
      Data_Definition_Mgt.pi_descending,
        -- requested property
      pv);
        -- value to be assigned

  -- close and bind DDef
  Data_Definition_Mgt.Close(dd);

  Text_Mgt.Set(name,"Dept_Salary_Index_DDef");
  Store_DDef(DDef => dd, name => name);
    -- Save created DDef under the symbolic name
    -- "Index_DDef"


end Create_dept_salary_DDef;


procedure Create_file_and_indexes(
    file_name:       System_Defs.text;
      -- New file's pathname.
    org_index_name:  System_Defs.text)
      -- Organization index's name.
  --
  -- Logic:
  --   Define descriptors for the file, the organization index,
  --   and the alternate index.  Create the file, build the
  --   organization index, and build the alternate index.
  --
  -- Note:
  --   You build the organization index built after creating
  --   the file, and the alternate index after creating the
  --   organiztion index.
  --
is
  new_file:  File_Defs.file_AD;
begin
  -- Create the file first.
  new_file :=  File_Admin.Create_file(
          name  => file_name,
          logical_file_descr => (
            -- Set the file's logical
            -- file descriptor.
              file_org       => File_Defs.unordered,
              DDef_specified => true,
              term_char      => File_Defs.term_char,
              record_DDef    => employee_DDef,
              record_layout  => (
                  DDef_specified => true),
              lock_escalation_count => 0,
              xm_locking            => true,
                -- Required for any record locking,
                -- including transaction locking.
              short_term_logging    => true,
                -- Required for transaction support.
              long_term_logging     => false,
              max_rec_num           =>
                  max_employee_count,
              bytes_per_bucket      => 4096,
              fill_factor           =>
                  File_Admin.fill_factor_dont_care,
              org_index             => org_index_name));

  -- Build the organization index for the file.
  File_Admin.Build_index(
      file => new_file,
      logical_index_descr => (
```

```
          -- Set the index descriptor for Department.
            name                => dept_index_name,
            active              => true,
            index_org           =>
                File_Defs.btree_index,
            duplicates_allowed => false,
            duplicate_order     =>
                File_Defs.by_increasing_record_ID,
            null_attribute      => File_Defs.none,
            DDef                => dept_index_DDef,
            phantom_protected   => false,
            utilization_maintenance => true,
            bytes_per_bucket    =>
                File_Defs.page_size));

    -- Build an alternate index for the file.
    File_Admin.Build_index(
        file => new_file,
            logical_index_descr => (
                name                    =>
                    dept_salary_index_name,
                active                  => true,
                index_org               =>
                    File_Defs.btree_index,
                  -- A unordered org index with
                  -- a b-tree index.
                duplicates_allowed      => false,
                duplicate_order         =>
                    File_Defs.by_increasing_record_ID,
                null_attribute          =>
                    File_Defs.none,
                DDef                    =>
                    dept_salary_index_DDef,
                phantom_protected       => true,
                  -- Uses bucket-level locking.
                utilization_maintenance => true,
                bytes_per_bucket        =>
                    File_Defs.page_size));

  end Create_file_and_indexes;

end Employee_Filing_Ex;
```

X-A.4.5 `Hello_ada_ex` Procedure

```
with Text_IO;

procedure Hello_ada_ex is
  --
  -- Function:
  --   Write "Hello, world!" on a separate line to the
  --   standard output, using Ada's "Text_IO" package.
begin
  Text_IO.Put_line("Hello, world!");
end Hello_ada_ex;
```

X-A.4.6 Hello_OS_ex Procedure

```
with Byte_Stream_AM,
     Device_Defs,
     Process_Mgt,
     Process_Mgt_Types,
     System;

procedure Hello_OS_ex is
  --
  -- Function:
  --   Write "Hello, world!" on a separate line to the
  --   standard output, using OS packages.

  hello:   constant string := "Hello, world!" & ASCII.LF;
  stdout:  constant Device_Defs.opened_device :=
      Process_Mgt.Get_process_globals_entry(
      Process_Mgt_Types.standard_output);
begin
  Byte_Stream_AM.Ops.Write(
      opened_dev => stdout,
      buffer_VA  => hello(1)'address,
      length     => System.ordinal(hello'length));
end Hello_OS_ex;
```

X-A.4.7 Join_File_Ex Package Specification

```
with Join_Interface,
     System;
package Join_File_Ex is
  --
  -- Function:
  --   This package provides examples using
  --   the DBMS support operations.
  --
  -- History:
  --   08-10-87, Paul Schwabe: initial revision.
  --   11-30-87, Paul Schwabe: update.
  --
  pragma external;

  -- Define some user buffer.
  --
  type stuff_buffer_type is
    array(1 .. 256) of character;

  -- Define local data structures.
  --
  type some_other_type is
    array(1 .. 256) of character;

  type user_info_type is
    record
      first_call:    boolean := true;
        -- This is reset by the user join procedure
        -- during the first call.
      comm_block:    Join_Interface.communication_block_VA;
        -- This is returned by the user join
        -- procedure.
      user_specific: some_other_type;
        -- Needed for the user's join algorithm.
    end record;

  function Join_ex(
      buffers_available:       System.ordinal;
        -- Number of 4kbyte file buffers reserved
        -- for this join.
      user_info:               System.address;
        -- Object for user process specific storage.
      records:                 Join_Interface.record_lists_AD)
        -- The list of record locations for each
        -- input device. Those are null the first time
        -- this routine is called.
    return Join_Interface.communication_block_VA;
             -- Contains the 'next block list' and the
             -- output buffers.
  pragma subprogram_value(Join_Interface.Block_join, Join_ex);
    --
    -- Function:
    --   The function Join_ex (subprogram type
    --   Join_Interface.Block_join) will be called
    --   from inside the Join_Interface.Join. (After
    --   having locked all the participating input
    --   devices on file level, we call the Join).


  procedure Join_call(
       num_input_devices: System.short_ordinal);
         -- Number of participating devices.
    -- Function:
    --   Calls the Join procedure.
    --

end Join_File_Ex;
```

X-A.4.8 Join_File_Ex Package Body

```ada
with Device_Defs,
     Join_Interface,
     System,
     Unchecked_Conversion;
package body Join_File_Ex is
  --
  -- Logic:
  --   This package body contains the implementations
  --   for the examples using the DBMS support
  --   operations.
  --

--
--                     UNCHECKED CONVERSIONS
--

  function Convert_comm_block_VA_to_address is
     new Unchecked_Conversion(
             source => Join_Interface.
                 communication_block_VA,
             target => System.address);

  function Convert_address_to_comm_block_VA is
     new Unchecked_Conversion(
             source => System.address,
             target => Join_Interface.
                            communication_block_VA);

  function Convert_address_to_next_block_VA is
     new Unchecked_Conversion(
             source => System.address,
             target => Join_Interface.
                            next_block_list_VA);

  function Convert_next_block_VA_to_address is
     new Unchecked_Conversion(
             source => Join_Interface.
                            next_block_list_VA,
             target => System.address);


-----------------------------------------------------------
--        BODY FOR THE SUBPROGRAM TYPE BLOCK_JOIN        --
-----------------------------------------------------------

  function Join_ex(
      buffers_available:  System.ordinal;
        -- Number of 4kbyte file buffers reserved
        -- for this join.
      user_info: System.address;
        -- Object for user specific storage.
      records:   Join_Interface.record_lists_AD)
        -- The list of record locations for each
        -- input device. Those are null the first time
        -- this routine is called.
    return Join_Interface.communication_block_VA
        -- Contains the 'next block list' and the
        -- output buffers.
    --
    -- Operation:
    --
  is
    u_info:        user_info_type;
      FOR u_info USE AT  user_info;
        -- Retypes the address to user_info_type.

    comm_block:    Join_Interface.communication_block;
      FOR comm_block USE AT
          Convert_comm_block_VA_to_address(
              u_info.comm_block);
        -- Just a rename.

    num_devices:   System.short_ordinal :=
                        records.num_devices;
```

```
        -- Number of input devices for this Join.

  begin

    -- First distribute the 'buffers_available' among
    -- the input devices in some manner. Make sure the
    -- number of buffers requested at a time does not
    -- exceed the numbers of buffers available.
    --
    -- .... lets say 2 buckets per block per input
    -- file is the result.


    if u_info.first_call then
      -- This is the first time this function is
      -- called. (This can also be recognized by
      -- checking the ADs in 'records', which are null
      -- at this time).

      for i in 1 .. num_devices loop
        -- Set up the communication block to condition
        -- Join for the next call.

        comm_block.position_blocks.next_blocks(i).
            block_size := 2;
          -- Two buckets per block.

        comm_block.position_blocks.next_blocks(i).
            position   := Join_Interface.next;
          -- We want to trace through the files from
          -- the beginning to the end. The Join will
          -- call this function the next time with
          -- record locations of those records
          -- contained in the first two buckets of the
          -- input file i. "Current" would deliver
          -- empty record location arrays at this
          -- stage. "Previous" would start with the
          -- last two buckets in the file.

      end loop;

    else
      -- This is not the first call to this function.

      -- Here is where a join algorithm takes place.
      --
      -- If i counts the devices from 1 ..
      -- num_devices, and if j counts the number of
      -- entries in one record_location_array (1 ..
      -- num_records), then the necessary data for the
      -- join algorithm can be retrieved
      -- via the following paths:

      -- num_records := records.rec_list_array(i).
      -- num_entries;
        -- Number of records per record location array.

      -- One record can be found in:
      --
      -- records.rec_list_array(i).rec_loc_array(j).
      --     record_VA
      -- records.rec_list_array(i).rec_loc_array(j).
      --     record_length
      -- records.rec_list_array(i).rec_loc_array(j).
      --     record_ID

      -- If the buckets scanned do not contain any
      -- records then the "number_of_entries" will be
      -- 0. It will be
      -- "Join_Interface.null_num_entries" when the
      -- end of the file has been exceeded.

      -- Now, join the records into the
      -- 'buffer_with_stuff'. . . . . . .

      -- Set up the comm_block with respect to the
      -- output buffers.
```

```
      --
      -- comm_block.out_buffers.output_length  :=
      -- some_value;
        --The length of the buffer contents
        -- in bytes. A non zero value provides for
        -- flushing the buffer to the output device.

      -- Set up the communication block with
      -- positioning information for the
      -- subsequent call:
      --
      -- comm_block.position_blocks(i).block_size := 2;
        -- Two buckets per block.

      -- comm_block.position_blocks(i).position   :=
      --     Join_Interface.next;
      -- Makes the Join call this
      -- function the next time with record locations
      -- of those records contained in the next two
      -- buckets of the input file i.
        null;

    end if;

    return u_info.comm_block;

  end Join_ex;


--------------------------------------------------------------
--                        THE CALL                          --
--------------------------------------------------------------

  -- The function Join_ex (subprogram type
  -- Join_Interface.Block_join) will be called from
  -- inside the Join_Interface.Join.

  -- (After having locked all the participating input
  -- devices on file level, we call the Join).


  procedure Join_call(
       num_input_devices:  System.short_ordinal)
         -- Number of participating devices.
    -- Operation:
    --   Calls the Join procedure.
    --
  is
    join_devices: Join_Interface.join_device_list(
                           num_input_devices);
      -- Input devices for the Join.

    out_file: Device_Defs.opened_device;
      -- Output rec_ID_stream device.

    buffer_reservation: Join_Interface.
                              buffer_reservation_block;
      -- Block which determines the number of buffers
      -- needed.

    u_info:     user_info_type;
      -- Global storage for the Block_join procedure.
      -- Will be passed to Block_join.

    comm_block: Join_Interface.communication_block;
      -- Instantiates the communication block.
      -- Contains the next_block list.

    buffer_with_stuff:   stuff_buffer_type;
      -- User records that will be copied to the output.

    length_of_one_stuff_record: constant
        System.ordinal := 8;
      -- Constant size of the "stuff records".
      -- the output buffers;

    next_blocks: Join_Interface.next_block_list(
```

```
                      num_entries => num_input_devices);
        -- The list that specifies which blocks to use
        -- for the next call.

   begin

      -- Hook the comm_block into user info.
      --
      u_info.comm_block :=
          Convert_address_to_comm_block_VA(
              comm_block'address);

      -- Initialize the comm_block.
      --
      comm_block.position_blocks :=
          Convert_address_to_next_block_VA(
              next_blocks'address);
        -- Unchecked conversion; see Ada-G.

      -- Set up the communication block with respect to
      -- the output buffers.
      --
      comm_block.out_buffers.output_buffer     :=
          buffer_with_stuff'address;
      comm_block.out_buffers.record_size       :=
          length_of_one_stuff_record;
      comm_block.out_buffers.alt_output_buffer :=
          System.null_address;
      comm_block.out_buffers.alt_record_size   := 0;
      --
      -- Here, the descriptors for the output buffers
      -- have to be set to make sure the buffers don't
      -- get flushed, since they do not contain any
      -- interesting data.
      --
      comm_block.out_buffers.output_length     := 0;
      comm_block.out_buffers.alt_output_length := 0;

      -- Get the ODOs for the input devices from somewhere.
      --
      -- join_devices := (. . .);

      -- Calculate how much buffers should be reserved
      -- by the Join at a time. Determine how many you
      -- need as a minimum; what's the optimal number?
      -- Do you want to wait until the buffers are
      -- available?
      --
      -- buffer_reservation := (...);

      -- Create and/or Open the output device
      --
      -- out_file := ....

      -- Initialize the user info.
      --
      -- u_info   := ....

      -- And off we go:
      --
      Join_Interface.Join(
           participating_devices => join_devices,
           buffers_to_reserve    => buffer_reservation,
           user_info             => u_info'address,
           join_procedure        =>
               Join_ex'subprogram_value,
           join_output           => out_file,
           alternate_output      => System.null_word);

   end Join_call;


end Join_File_Ex;
```

X-A.4.9 Record_Locking_Ex Package Specification

```
with Device_Defs,
     System_Defs;
package Record_Locking_Ex is
  --
  -- Function:
  --   This package contains the examples for
  --   using the record locking in your
  --   applications.
  --
  -- History:
  --   01-07-88, Paul Schwabe: initial version.
  --
  pragma external;

  procedure Level_3_update(
      file_name: System_Defs.text);
    --
    -- Function:
    --   This example is designed to illustrate level
    --   3 consistency.  It reads the employee records
    --   in a key range and updates the salaries.
    --
    --   Does an index-sequential read of an
    --   unordered file using a single b-tree alternate
    --   index.  The read call uses a "write" lock mode
    --   because the record will be updated after the read.
    --

end Record_Locking_Ex;
```

X-A.4.10 `Record_Locking_Ex` Package Body

```
with Device_Defs,
     Employee_Filing_Ex,
     File_Admin,
     File_Defs,
     Record_AM,
     System,
     System_Defs,
     Text_Mgt,
     Transaction_Mgt;

use System;

package body Record_Locking_Ex is
  --
  -- Logic:
  --   This package body contains the
  --   the implementations for the record
  --   locking examples.
  --
  buffer:  string(1 .. integer(
    Employee_Filing_Ex.max_rec_size));
    -- Buffer is large enough to hold any employee
    -- record.

  current_record_addr:  constant
      System.address := buffer'address;
  current_record_VA:  constant
      Employee_Filing_Ex.employee_record_VA :=
      Employee_Filing_Ex.Employee_record_VA_from_VA(
          current_record_addr);

  bytes_read:  System.ordinal;
    -- Number of bytes in current record.


  procedure Level_3_update(
      file_name: System_Defs.text)
        -- An opened device for transaction T1, opened
        -- for input on an employee file.
    --
    -- Operation:
    --   Reads all records in a relative file and
    --   totals the salaries.
    --
    --   Does an index-sequential read of an
    --   unordered file using a single b-tree alternate
    --   index.  Transaction T1 (a reader) reads
    --   employee records using the write_lock lock
    --   mode, locking the file from other readers and
    --   writers.
    --
  is
    opened_file: Device_Defs.opened_device;

    total_salary: Employee_Filing_Ex.monthly_salary
        := 0.00;

    start_key_value:  constant Employee_Filing_Ex.
        dept_salary_key_buffer := (
            dept      => 100,
              -- Lowest department, ascending.
            salary    => 10_000.00);
              --Highest salary, descending.

    stop_key_value:   constant Employee_Filing_Ex.
        dept_salary_key_buffer := (
            dept      => 500,
              -- Highest department, ascending.
            salary    => 1_000.00);
              -- Lowest salary, descending.

    level_3_mode: Record_AM.open_mode_value(Record_AM.level_3) :=
        (mode_id => Record_AM.level_3,
         value   => true);
```

```
  begin
    Transaction_Mgt.Start_transaction;
      -- Started on behalf of transaction T1,
      -- the level 3 reader.
      -- Any updates, deletes or inserts
      -- (not shown) within this transaction
      -- can be rolled back if
      -- the transaction aborts.

    opened_file := Record_AM.Open_by_name(
        name         =>  file_name,
        input_output =>  Device_Defs.inout,
        allow        =>  Device_Defs.anything);

    Record_AM.Ops.Set_open_mode(
        opened_dev => opened_file,
        mode_value => level_3_mode);
      -- Sets level 3 consistency.

    Record_AM.Keyed_Ops.Set_key_range(
       opened_file,
       index => Employee_Filing_Ex.
           dept_salary_index_name,
       select_range => (
           start_comparison =>
               Record_AM.inclusive,
           start_value      => (
               start_key_value'address,
               start_key_value'size / 8),
           stop_comparison  =>
               Record_AM.inclusive,
           stop_value       => (
               stop_key_value'address,
               stop_key_value'size / 8)));

    loop
      bytes_read := Record_AM.Ops.Read(
          opened_dev => opened_file,
          buffer_VA  => current_record_addr,
          length     => Employee_Filing_Ex.
              max_rec_size,
          lock       => Record_AM.write_lock,
          unlock     => Record_AM.no_unlock);
      -- Another caller cannot read or update
      -- the same record at any time.

      if current_record_VA.salary = 3_000.00 then
         current_record_VA.salary :=
             current_record_VA.salary + 300.00;

         Record_AM.Ops.Update(
             opened_dev => opened_file,
             modifier   => Record_AM.current,
             buffer_VA  => current_record_addr,
             length     => Employee_Filing_Ex.
                 max_rec_size,
             timeout    => Record_AM.wait_forever,
             status     => null);
       end if;
    end loop;

    exception
      when Device_Defs.end_of_file =>
        Transaction_Mgt.Commit_transaction;
          -- Everthing's OK.

      when others =>
          -- Something's bad.
        null;
  end Level_3_update;

end Record_Locking_Ex;
```

X-A.4.11 Output_bytes_ex Procedure

```
with Byte_Stream_AM,
     Device_Defs,
     Process_Mgt,
     Process_Mgt_Types,
     System,
     System_Defs,
     Unchecked_conversion;

  procedure Output_bytes_ex(
      name:  System_Defs.text)
        -- Input device to read.
    --
    -- Function:
    --   Opens the named input device and
    --   copies bytes from it to the caller's
    --   standard output, until end-of-file.
  is
    source_opened_device:  Device_Defs.opened_device;
    dest_opened_device:    Device_Defs.opened_device;
    function Opened_device_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Device_Defs.opened_device);
    BUFSIZE:     constant System.ordinal := 4_096;
    buffer:      array(1 .. BUFSIZE) of
                 System.byte_ordinal;
    bytes_read:  System.ordinal;
begin
  source_opened_device :=
      Byte_Stream_AM.Open_by_name(
          name         => name,
          input_output => Device_Defs.input,
          allow        => Device_Defs.readers);
  dest_opened_device := Opened_device_from_untyped(
      Process_Mgt.Get_process_globals_entry(
          Process_Mgt_Types.standard_output));

    loop
      bytes_read := Byte_Stream_AM.Ops.Read(
          source_opened_device,
          buffer'address,
          BUFSIZE);
      Byte_Stream_AM.Ops.Write(
          dest_opened_device,
          buffer'address,
          bytes_read);
    end loop;
  exception
    when Device_Defs.end_of_file =>
      Byte_Stream_AM.Ops.Close(
          source_opened_device);
end Output_bytes_ex;
```

X-A.4.12 Output_records_ex Procedure

```
with Device_Defs,
     Object_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     Record_AM,
     System,
     System_Defs,
     Unchecked_conversion;

procedure Output_records_ex(
    name:  System_Defs.text)
      -- Pathname of device.  Caller must have
      -- read rights.
  --
  -- Operation:
  --   Opens a named device, reads a stream
  --   of records, and writes the records to
  --   the caller's standard output, until
  --   end-of-file.
  --
  -- Notes:
  --   The record buffer is dynamically sized
  --   so that records of any length can be
  --   handled.  Recovery from buffer overflow
  --   uses the "rest_of_current" rather than
  --   "current" read option, because some
  --   devices, such as pipes, do not support
  --   the "current" option.
  --
  -- Exceptions:
  --   Device_Defs.device_in_use -
  --     The device is being used by
  --     an application that does not
  --     allow concurrent readers.
  --   Device_Defs.open_mode_conflict -
  --     The named object does not
  --     allow opens for input.
  --   Device_Defs.device_inconsistent
  --   Device_Defs.device_offline
  --   Device_Defs.device_inoperative
  --   Device_Defs.transfer_error
  --   Directory_Mgt.no_access -
  --     There is no such pathname
  --     or the caller does not have
  --     access to the named device.
  --   Directory_Mgt.name_too_long -
  --     The pathname or some part of it
  --     exceeds an OS size limit.
  --   File_Defs.volume_space_exhausted
  --   Record_AM.XXX -
  --     Many "Record_AM" exceptions
  --     can be raised.  See "Read" and
  --     "Insert" in "Record_AM.Ops".
is
  use System;  -- Import ordinal operators.
  source_opened_device:  Device_Defs.opened_device;
  dest_opened_device:    Device_Defs.opened_device;
  buffer_size:  System.ordinal := 256;
  buffer_AD:  System.untyped_word :=
      Object_Mgt.Allocate(buffer_size/4);
    -- 64 words (256 bytes) is the initial buffer
    -- size.  Buffer size is increased as needed.
    -- The buffer is in a separate object for easy
    -- resizing.
  bytes_read:       System.ordinal := 0;
    -- If record requires multiple "Read" calls,
    -- then this variable tracks bytes read so far.
  read_status_VA:  Record_AM.operation_status_VA :=
      new Record_AM.operation_status_record;
  read_position:   Record_AM.position_modifier :=
      Record_AM.next;
    -- If record requires multiple "Read" calls,
    -- then this variable is assigned
    -- "Record_AM.rest_of_current" for the
```

```
    -- 2nd through Nth reads.
  function Opened_device_from_untyped is new
      Unchecked_conversion(
          source => System.untyped_word,
          target => Device_Defs.opened_device);
begin
  source_opened_device := Record_AM.Open_by_name(
      name         => name,
      input_output => Device_Defs.input,
      allow        => Device_Defs.readers);
  dest_opened_device := Opened_device_from_untyped(
      Process_Mgt.Get_process_globals_entry(
          Process_Mgt_Types.standard_output));

  loop

    loop
      begin
        bytes_read := bytes_read +
            Record_AM.Ops.Read(
                source_opened_device,
                read_position,
                System.address'(
                    bytes_read,
                    buffer_AD),
                buffer_size - bytes_read,
                status => read_status_VA);

        -- When control reaches this point, "Read"
        -- succeeded without a length error and
        -- this loop can be exited.
        EXIT;

      exception
        when Device_Defs.length_error =>
          buffer_size := read_status_VA.rec_length;
          if buffer_size =
              Record_AM.unknown_length then
            buffer_size := 2 * 4 *
                Object_Mgt.Get_object_size(buffer_AD);
              -- Double the buffer size if an exact
              -- new size is not available.
          end if;
          Object_Mgt.Resize(
              buffer_AD,
              (buffer_size+3)/4);
            -- May make object even bigger than
            -- requested, but that's OK.
          read_position := Record_AM.rest_of_current;
      end;
    end loop;

    Record_AM.Ops.Insert(
        dest_opened_device,
        System.address'(0, buffer_AD),
        bytes_read);

    -- Reset variables to read the next record
    -- into the beginning of the buffer:
    --
    bytes_read := 0;
    read_position := Record_AM.next;
  end loop;

exception
  when Device_Defs.end_of_file =>
    Record_AM.Ops.Close(source_opened_device);
end Output_records_ex;
```

X-A.4.13 `Print_cmd_ex` Procedure

```
with Byte_Stream_AM,
     CL_Defs,
     Command_Handler,
     Device_Defs,
     Directory_Mgt,
     Print_Cmd_Messages,  -- Message package.
     Incident_Defs,
     Message_Services,
     Process_Mgt,
     Process_Mgt_Types,
     Spool_Defs,
     Spool_Device_Mgt,
     String_List_Mgt,
     System,
     System_Defs,
     Text_Mgt;

procedure Print_cmd_ex
  --
  -- Function:
  --   Defines a command to print from a file or other
  --   byte stream source
  --
  -- Command Definition:
  --   The command has the form:
  --
  --     print
  --         [source=<pathname>]
  --         [on=<pathname>]
  --
  -- The on argument can either be a spool queue or a
  -- printer (for direct printing).  The default is a
  -- system standard spooling device.  The source
  -- argument will default to standard input.
  --
  --*C* set.message_file :file = \
  --*C*     /examples/msg/example_messages
  --*C*
  --*C* create.command :cmd_def = print.inv_cmd \
  --*C*     :cmd_name = print
  --*C*
  --*C*   define.argument source
  --*C*       :type = string
  --*C*       set.lexical_class symbolic_name
  --*C*       set.maximum_length 252
  --*C*       set.value_default ""
  --*C*     end
  --*C*
  --*C*   define.argument on
  --*C*       :type = string
  --*C*       set.lexical_class symbolic_name
  --*C*       set.maximum_length 80
  --*C*       set.value_default ""
  --*C*     end
  --*C* end
  --*C*
  --*C* run "store.command_definitions \
  --*C*     :program = print \
  --*C*     :invocation_cmd = print.inv_cmd"
  --*C*
  --*C* run "store.default_message_file \
  --*C*     print \
  --*C*     print.msg"

is

  use System;


  opened_cmd:        Device_Defs.opened_device;
                       -- Opened command input device.

  -- source variables
  source:            System_Defs.text(252);
```

```
                        -- Pathname of file or device
                        -- print from

  open_source:      Device_Defs.opened_device;

  -- "on" variables
  on_device:        System_Defs.text(Incident_Defs.txt_length);
                        -- Pathname of spool queue or
                        -- printer

  on_untyped:       System.untyped_word;

  spool_queue:      Device_Defs.device;

  print_device:     Device_Defs.device;

  no_print_device:  exception;

  sheet_size:       constant Spool_Defs.size_t :=
      (132,66);

  open_print:       Device_Defs.opened_device;

  -- buffer variables
  buffer_size:      constant System.ordinal := 4_096;
  buffer:           array(1..buffer_size) of
      System.byte_ordinal;
  bytes_read:       System.ordinal;

begin

  -- Get command arguments:
  --
  opened_cmd :=
      Command_Handler.
      Open_invocation_command_processing;
  Command_Handler.Get_string(opened_cmd, 1,
      arg_value => source);
  Command_Handler.Get_string(opened_cmd, 2,
      arg_value => on_device);
  Command_Handler.Close(opened_cmd);

  -- assign defaults if parameter was not specified

  if source.length = 0 then
    open_source :=
        Process_Mgt.Get_process_globals_entry(
        Process_Mgt_Types.standard_input);
       -- standard input from terminal
  else
    open_source := Byte_Stream_AM.Open_by_name(
        name         => source,
        input_output => Device_Defs.input);
  end if;

  if on_device.length = 0 then
    Text_Mgt.Set(on_device,"/dev/lpq");
    -- Correct name of default system spool queue is
    -- TBD
  end if;

  -- check the "on_device" for spooled or direct
  -- printing, else error

  on_untyped := Directory_Mgt.Retrieve(on_device);
  if Spool_Defs.Is_spool_queue(on_untyped) then
    print_device :=
        Spool_Device_Mgt.Create_print_device(
            spool_queue => spool_queue,
            pixel_units => false,
            print_area  => sheet_size);

  elsif Spool_Defs.Is_print_device(on_untyped) then
    print_device :=
        Spool_Device_Mgt.Create_print_device(
            spool_queue => spool_queue,
            pixel_units => false,
```

```
            print_area  => sheet_size,
            print_mode  => Spool_Defs.page_wise);
              -- direct printing

  else
    RAISE no_print_device;
  end if;

  open_print := Byte_stream_AM.Ops.Open(
                           print_device,
                           Device_Defs.output);

  while not
      Byte_Stream_AM.Ops.At_end_of_file(open_source)
      loop
    bytes_read := Byte_Stream_AM.Ops.Read(
        opened_dev => open_source,
        buffer_VA  => buffer'address,
        length     => buffer_size);

    Byte_Stream_AM.Ops.Write(
        opened_dev => open_print,
        buffer_VA  => buffer'address,
        length     => bytes_read);
   end loop;

  Byte_Stream_AM.Ops.Close(open_source);
  Byte_Stream_AM.Ops.Close(open_print);

  exception
    when no_print_device =>
      Message_Services.Write_msg(
          Print_Cmd_Messages.no_print_device_code,
          Incident_Defs.message_parameter(
              typ => Incident_Defs.txt,
              len => on_device.max_length)'(
              typ => Incident_Defs.txt,
              len => on_device.max_length,
              txt_val => on_device));

    when Spool_Device_Mgt.units_not_supported =>
       Message_Services.Write_msg(
            Print_Cmd_Messages
            .units_not_supported_code,
            Incident_Defs.message_parameter(
                typ => Incident_Defs.txt,
                len => on_device.max_length)'(
                typ => Incident_Defs.txt,
                len => on_device.max_length,
                txt_val => on_device));
end Print_cmd_ex;
```

X-A.4.14 Print_Cmd_Messages Package

```
with Incident_Defs,
     System,
     System_Defs;

package Print_Cmd_Messages is
  --
  -- Function:
  --   Define messages used by Print_cmd_ex
  --   All messages defined use a module ID of 0.

  print_msg_pathname:  constant System_Defs.text_AD :=
      new System_Defs.text'(
          32,32,"/examples/msg/print_cmd_messages");
    -- AD to pathname of message file, bound to
    -- "msg_obj", following.
    -- *This will go away when "pragma bind" changes.*

  msg_obj:  constant System.untyped_word :=
      System.null_word;
    pragma bind(msg_obj,
                "example_messages.print_msg_pathname");
    -- Message object for incident codes in
    -- example programs, bound to above
    -- "message_file_pathname".
    --
    -- *When the resident compiler and linker are*
    -- *ready, this pragma will become:*
    -- |  pragma bind(msg_obj,
    -- |           "/examples/msg/print_cmd_messages");


  no_print_device_code:
      constant Incident_Defs.incident_code :=
      (0, 1, Incident_Defs.information, msg_obj);
  --
  --*M* store :module=0 :number=1 \
  --*M*     :msg_name=name_space_created_code \
  --*M*     :short = \
  --*M*     "Print Device $p1<on> does not exist."

  units_not_supported_code:
      constant Incident_Defs.incident_code :=
      (0, 2, Incident_Defs.information, msg_obj);
  --
  --*M* store :module=0 :number=2 \
  --*M*     :msg_name=units_not_supported_code \
  --*M*     :short = \
  --*M*     "Unit $p1<on> not supported."

end Print_Cmd_Messages;
```

X-A.4.15 Record_AM_Ex Package Specification

```
with Device_Defs,
     Employee_Filing_Ex,
     Record_AM,
     System,
     System_Defs;

package Record_AM_Ex is
  --
  -- Function:
  --   This package contains the example subprograms
  --   for using the Record_AM package.
  --
  -- History:
  --   08-10-87, Paul Schwabe: initial version.
  --   11-23-87, Paul Schwabe: revision.
  --
  pragma external;

  function Get_record_ID(
      opened_file:  Device_Defs.opened_device)
        -- An opened device, opened for input on an
        -- employee file.
    return Record_AM.record_ID;
    --
    -- Operation:
    --   Returns a record ID from the operation status
    --   information. The record ID can be used in
    --   subsequent retrieval operations to maximize
    --   access time to the specified record.


  function Get_record_number(
      opened_file:  Device_Defs.opened_device)
        -- An opened device, opened for input on an
        -- employee file.
    return System.ordinal;
    --
    -- Operation:
    --   Returns a record number from the operation
    --   status information.  The record number can be
    --   used in subsequent retrieval operations for
    --   relative files.


  procedure Insert_record(
      opened_file: Device_Defs.opened_device);
       -- An opened device, opened for input on an
       -- employee file.
    --
    -- Function:
    --   Inserts a record into a structured file.
    --
    --   Applicable for any file organization.
    --   Position of the inserted record in the file
    --   is determined by the system.  The new record
    --   is automatically assigned a record ID.


  procedure Read_random_by_record_ID(
      opened_file:  Device_Defs.opened_device;
      rec_id:       Record_AM.record_ID);
        -- An opened device, opened for input on an
        -- employee file.
    --
    -- Function:
    --   Reads a record randomly using a previously
    --   retrieved record ID from the operation status
    --   information. This is the fastest possible
    --   random access to a record using any
    --   structured file organization.


  procedure Read_random_by_record_number(
      opened_file:  Device_Defs.opened_device;
```

```
        rec_number:   System.ordinal);
          -- An opened device, opened for input on an
          -- employee file.
      --
      -- Function:
      --   Reads a record randomly from a relative file
      --   using a previously retrieved record ID from
      --   the operation status information.  Record
      --   numbers are only applicable for relative
      --   files.



    procedure Read_next_simple_index(
        opened_file:  Device_Defs.opened_device);
          -- An opened device, opened for input on an employee
          -- file.
      --
      -- Function:
      --   Reads a range of records in the "Dept" index.
      --
      --   Positions to the beginning of the range and
      --   reads successive records until the end.  The
      --   start value is to the left of the index.
      --   This composite index is read by ascending key
      --   values starting at the lowest key value in
      --   the range.
      --
      --   Dept (asc)  A  B  ...  X   Y
      --           --->                EOF
      --   The position_modifier value is Record_AM.next
      --
      -- Notes:
      --   This function replaces any previous key range
      --   and changes the file's record pointer.
      --
      --   The "Dept" index is ascending on department.
      --   Returns all employee records for the
      --   departments in the specified range.
      --



    procedure Read_prior_simple_index(
        opened_file:  Device_Defs.opened_device);
          --   An opened device, opened for input on an
          --   employee file.
      --
      -- Function:
      --   Reads a range of records in the "Dept" index.
      --
      --   Positions to the end of the range and reads
      --   successive records until the beginning.  The
      --   start value is to the right of the index.
      --   This composite index is read by ascending key
      --   values starting at the lowest key value in
      --   the range.
      --
      --   Dept (asc)  A  B  ...  X  Y
      --            EOF             <---
      --   The position_modifier value is
      --   Record_AM.prior
      --
      -- Notes:
      --   This function replaces any previous key range
      --   and changes the file's record pointer.
      --
      --   The "Dept" index is ascending on department.
      --   Returns all employee records for the
      --   departments in the specified range.
      --



    procedure Read_duplicates(
        opened_file:  Device_Defs.opened_device);
          --   An opened device, opened for input on an
```

```
        --   employee file.
    --
    -- Function:
    --   Reads a duplicate records in the specified
    --   "Dept" index.
    --
    --   Positions to the specified record and reads
    --   all duplicates until the end.
    --
    --     Dept (asc)   A  A  ...    A  A
    --               --->                EOF
    --   The position_modifier value is Record_AM.next
    --
    -- Notes:
    --   This function replaces any previous key range
    --   and changes the file's record pointer.
    --
    --   The "Dept" index is ascending on department.
    --   Returns all employee records for the
    --   departments in the specified range.
    --
    --   The range contains employees in "Accounting"
    --   through "Marketing".
    --
    --   If the "Dept" index were specified as
    --   non-unique, returns duplicate recores for a
    --   particular "Dept" key value.  For example,
    --   one record might contain fields on
    --   management, cost control, and history. A
    --   second record might simply hold text.
    --


procedure Delete_records_sequential(
    opened_file:  Device_Defs.opened_device);
        --   An opened device, opened for input on an
        --   employee file.
    --
    -- Function:
    --   Deletes a range of records using the
    --   department name as a key.  This example shows
    --   that a Read or Set_position is not required
    --   to preface each Delete.  The current record
    --   pointer advances after each Delete.

procedure Read_and_update_by_key(
    opened_file:  Device_Defs.opened_device);
        --   An opened device, opened for input on an
        --   employee file.
    --
    -- Function:
    --   Updates a record within a range of records.
    --   This example shows that the current record
    --   pointer does NOT advance after the
    --   Update_by_key.



procedure Read_records_reverse_sequential(
    opened_file:  Device_Defs.opened_device);
        --   An opened device, opened for input on an
        --   employee file.
    --
    -- Function:
    --   Reads all records in a reverse sequence.
    --   Shows Shows physical-sequential access.
    --
    --   Positions to the end of the sequence and
    --   reads successive records until the beginning.
    --   After each read, the current record pointer
    --   is positioned to the prior record.


procedure Read_records_sequential(
    opened_file:  Device_Defs.opened_device);
        --   An opened device, opened for input on an
        --   employee file.
```

```
   --
   -- Function:
   --   Reads all records in a sequence.  Shows
   --   physical-sequential access.
   --
   --   Positions to the start of the sequence and
   --   reads successive records until the end.
   --
   -- Notes:
   --   Advances the file's current record pointer
   --   forward after each read.
   --


  procedure Read_and_delete_records(
      opened_file:  Device_Defs.opened_device);
        --   An opened device, opened for input on an
        --   employee file.
   --
   -- Function:
   --   Reads and deletes selected records in a
   --   sequence.
   --
   --   Positions to the beginning of the sequence
   --   and reads successive records until the end.
   --   After each read, a record is checked and then
   --   deleted if it satisfies the specified
   --   conditions.  The current record pointer is
   --   positioned to the next record after the
   --   deleted record.


  procedure Read_and_update_records(
      opened_file:  Device_Defs.opened_device);
        --   An opened device, opened for input on an
        --   employee file.
   --
   -- Function:
   --   Reads and updates records in a sequence.
   --
   --   Positions to the beginning of the sequence
   --   and reads successive records until the end.
   --   After each read, the current record pointer
   --   is positioned to the next record.


  procedure Update_salary_example(
      T2_opened_file:  Device_Defs.opened_device);
        --   An opened device for transaction T1,
        --   opened for input on an employee file.
   --
   -- Function:
   --   Does an index-random update of a record in an
   --   indexed relative file.
   --
   --   The Update_salary_example procedure starts
   --   transaction T2 to double an employee's
   --   salary.  If transaction T2 aborts, then the
   --   update is rolled back.
   --
   -- Notes:
   --   The example relative file is created with the
   --   following parameters:
   --      xm_locking => true
   --      short_term_logging => true
   --   The example index (with a key built on
   --   "employee ID") is built with
   --   phantom_protected => false.
   --

end Record_AM_Ex;
```

X-A.4.16 Record_AM_Ex Package Body

```
with Device_Defs,
     Employee_Filing_Ex,
     File_Admin,
     File_Defs,
     Record_AM,
     System,
     System_Defs,
     Transaction_Mgt;

 -- For Importing operations.
use Employee_Filing_Ex,
    System,
    System_Defs;

package body Record_AM_Ex is
  --
  -- Logic:
  --   Provides the implementation code for the
  --   Record_AM examples.
  --

  --
  -- CONSTANT AND VARIABLE DECLARATIONS
  --

  buffer: string(1 .. integer(Employee_Filing_EX.max_rec_size));
    -- Buffer is large enough to hold any employee
    -- record.

  current_record_addr:  constant System.address :=
      buffer'address;
  current_record_VA:  constant Employee_Filing_EX.
      employee_record_VA := Employee_Filing_EX.
          Employee_record_VA_from_VA(
              current_record_addr);

  pay_raise:  constant float := 2.0;

  bytes_read:  System.ordinal;
    -- Number of bytes in current record.

  read_status_VA:  Record_AM.operation_status_VA :=
      new Record_AM.operation_status_record;
    -- Virtual address of status record.

  -- Employee name constant.
  employee:  constant Employee_Filing_EX.person_name :=
      (Employee_Filing_EX.max_text_length,
       10,
       "Einstein, Albert         ");

  --
  -- SUBPROGRAM DECLARATIONS
  --

  function Get_record_ID(
      opened_file:  Device_Defs.opened_device)
    -- An opened device, opened for input on an
    -- employee file.
    return Record_AM.record_ID
    -- Note:
    --   Records in any structured file can have
    --   record IDs, but only records in relative
    --   files can have record numbers!
  is
  begin
    Record_AM.Ops.Set_position(
        opened_dev => opened_file,
        where =>      Record_AM.record_specifier(
            type_of_specifier => Record_AM.first)'(
                type_of_specifier => Record_AM.first));
    loop
      bytes_read := Record_AM.Ops.Read(
          opened_dev => opened_file,
```

```
        buffer_VA  => buffer'address,
        length     => buffer'length,
        status     => read_status_VA);
    if current_record_VA.name = employee then
      RETURN read_status_VA.rec_ID;

    end if;
  end loop;

exception
  when Device_Defs.end_of_file =>
    RETURN Record_AM.null_record_ID;

end Get_record_ID;


function Get_record_number(
    opened_file:  Device_Defs.opened_device)
       -- An opened device, opened for input on an
       -- employee file.
  return System.ordinal
  --
is
begin
  Record_AM.Ops.Set_position(
      opened_dev => opened_file,
      where =>      Record_AM.record_specifier(
          type_of_specifier => Record_AM.first)'(
              type_of_specifier => Record_AM.first));
  loop
    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length,
        status     => read_status_VA);
    if current_record_VA.name = employee then
      RETURN read_status_VA.rec_num;

    end if;
  end loop;

exception
  when Device_Defs.end_of_file =>
    RETURN 0;

end Get_record_number;



procedure Insert_record(
    opened_file: Device_Defs.opened_device)
     -- An opened device, opened for input on an
     -- employee file.
  --
is
begin
  -- Obtain the new record from
  -- somewhere (form or file)
  -- and load the record buffer.

  Record_AM.Ops.Insert(
      opened_dev => opened_file,
      buffer_VA  => buffer'address,
      length     => System.ordinal(
          Employee_Filing_EX.max_rec_size));

end Insert_record;


procedure Read_random_by_record_ID(
    opened_file:  Device_Defs.opened_device;
    rec_ID:       Record_AM.record_ID)
       -- An opened device, opened for input on an
      -- employee file.
is
begin
  Record_AM.Ops.Set_position(
```

```
        opened_file,
        where =>      Record_AM.record_specifier(
            type_of_specifier => Record_AM.id)'(
                type_of_specifier => Record_AM.id,
                rec_id            => rec_ID));

    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length);

  end Read_random_by_record_ID;


  procedure Read_random_by_record_number(
      opened_file:  Device_Defs.opened_device;
      rec_number:   System.ordinal)
        -- An opened device, opened for input on an
        -- employee file.
  is
  begin
    Record_AM.Ops.Set_position(
        opened_file,
        where =>      Record_AM.record_specifier(
            type_of_specifier => Record_AM.number)'(
                type_of_specifier => Record_AM.number,
                rec_num           => rec_number));
    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length);

  end Read_random_by_record_number;


  procedure Read_next_simple_index(
      opened_file:  Device_Defs.opened_device)
        -- An opened device, opened for input on an
        -- employee file.
    --
  is
    start_key_value:  constant Employee_Filing_EX.
        dept_key_buffer := (dept => 100);
         -- Lowest deptartment for
         -- ascending key field.

    start_key_descr:  constant
      Record_AM.key_value_descr := (
        start_key_value'address,
        start_key_value'size / 8);

    stop_key_value:   constant Employee_Filing_EX.
       dept_key_buffer := (dept => 500);
        -- High end for ascending key field.

    stop_key_descr:  constant
      Record_AM.key_value_descr := (
        stop_key_value'address,
        stop_key_value'size / 8);
  begin
    Record_AM.Keyed_Ops.Set_key_range(
        opened_dev   => opened_file,
        index        =>
           Employee_Filing_EX.dept_index_name,
        select_range => (
            start_comparison => Record_AM.exclusive,
            start_value      => start_key_descr,
            stop_comparison  => Record_AM.inclusive,
            stop_value       => stop_key_descr));

    loop
      bytes_read := Record_AM.Ops.Read(
          opened_dev => opened_file,
          modifier   => Record_AM.next,
            -- Next is normally defaulted.
          buffer_VA  => buffer'address,
          length     => buffer'length);
```

```

      -- DO ANY NEEDED PROCESSING HERE.

    end loop;

  exception
    when Device_Defs.end_of_file =>
      null;
  end Read_next_simple_index;


  procedure Read_prior_simple_index(
      opened_file:  Device_Defs.opened_device)
        -- An opened device, opened for input on an
        -- employee file.
    --
  is
    start_key_value:  constant Employee_Filing_EX.
        dept_key_buffer := (dept => 500);
         -- High end for ascending key field.

      start_key_descr:  constant
      Record_AM.key_value_descr := (
        start_key_value'address,
        start_key_value'size / 8);

    stop_key_value:   constant Employee_Filing_EX.
        dept_key_buffer := (dept => 100);
        -- Lowest department for
        -- ascending key field.

    stop_key_descr:  constant
      Record_AM.key_value_descr := (
        stop_key_value'address,
        stop_key_value'size / 8);

  begin
    Record_AM.Keyed_Ops.Set_key_range(
        opened_dev => opened_file,
        index      =>
          Employee_Filing_EX.dept_index_name,
        select_range => (
            start_comparison => Record_AM.exclusive,
            start_value      => start_key_descr,
            stop_comparison  => Record_AM.inclusive,
            stop_value       => stop_key_descr));

    loop
      bytes_read := Record_AM.Ops.Read(
          opened_dev => opened_file,
          modifier   => Record_AM.prior,
            -- Sets read modifier to prior.
          buffer_VA  => buffer'address,
          length     => buffer'length);

      -- DO ANY NEEDED PROCESSING HERE.

    end loop;
  exception
    when Device_Defs.end_of_file =>
      null;
  end Read_prior_simple_index;



  procedure Read_duplicates(
      opened_file:  Device_Defs.opened_device)
        -- An opened device, opened for input on an
        -- employee file.
  is
    start_key_value:  constant Employee_Filing_EX.
        dept_key_buffer := (dept => 305);
           -- Start value for duplicate
           -- key field.


    start_key_descr:  constant Record_AM.
```

```
        key_value_descr := (
            start_key_value'address,
            start_key_value'size / 8);

    stop_key_value:   constant Employee_Filing_EX.
       dept_key_buffer := (dept => 305);
          -- Stop value for duplicate
          -- key field.


    stop_key_descr:  constant Record_AM.
        key_value_descr := (
            stop_key_value'address,
             stop_key_value'size / 8);
  begin
    Record_AM.Keyed_Ops.Set_key_range(
        opened_dev    => opened_file,
        index         => Employee_Filing_EX.
            dept_index_name,
        select_range => (
            start_comparison => Record_AM.inclusive,
            start_value      => start_key_descr,
            stop_comparison  => Record_AM.inclusive,
            stop_value       => stop_key_descr));
    loop
      bytes_read := Record_AM.Ops.Read(
          opened_dev => opened_file,
          modifier   => Record_AM.next,
            -- Normally defaulted.
          buffer_VA  => buffer'address,
          length     => buffer'length);

      -- DO ANY PROCESSING HERE

    end loop;
  exception
    when Device_Defs.end_of_file =>
      null;

  end Read_duplicates;

  procedure Delete_records_sequential(
      opened_file:  Device_Defs.opened_device)
        -- An opened device, opened for input on an
        -- employee file.
    -- Logic:
    --  Do a Set_key_range for a range of departments
    --  to delete.  Set up a loop for the deletes with
    --  the position_modifer = current.  (Key point: a
    --  Read or Set_position is not required to
    --  preface each Delete in the loop.  The current
    --  record pointer advances after each Delete)

  is
    start_key_value:  constant Employee_Filing_EX.
        dept_key_buffer := (dept  => 150);
           -- Low end for ascending key field.

    start_key_descr: constant Record_AM.
        key_value_descr := (
            start_key_value'address,
            start_key_value'size / 8);

    stop_key_value: constant Employee_Filing_EX.
        dept_key_buffer := (dept => 200);
          -- High end for ascending
          -- key field.

    stop_key_descr:  constant Record_AM.
        key_value_descr := (
            stop_key_value'address,
            stop_key_value'size / 8);
  begin
    Record_AM.Keyed_Ops.Set_key_range(
        opened_dev    => opened_file,
        index         => Employee_Filing_EX.
            dept_index_name,
```

```
          select_range => (
              start_comparison => Record_AM.inclusive,
              start_value      => start_key_descr,
              stop_comparison  => Record_AM.inclusive,
              stop_value       => stop_key_descr));
    loop
      -- CRP is updated after each delete
      -- (no read is necessary to preface
      -- the Delete).
      Record_AM.Ops.Delete(
          opened_dev => opened_file,
          modifier   => Record_AM.current,
            -- Normally defaulted.
          timeout    => Record_AM.wait_forever,
          status     => null);

    end loop;

  exception
    when Device_Defs.end_of_file =>
      null;

  end Delete_records_sequential;


  procedure Read_and_update_by_key(
      opened_file:  Device_Defs.opened_device)
        -- An opened device, opened for input on an
        -- employee file.
    --
    -- Logic:
    --  Do a Set_key_range for a range of departments
    --  to update.  Set up a read loop using
    --  position_modifier = next.  Do a comparison
    --  to trap a record to update.  When rec_in =
    --  record_of_interest, do an Update_by_key.
    --  (Key point: the current record pointer does
    --  NOT advance after the Update_by_key.)
  is
    start_key_value:  constant Employee_Filing_EX.
        dept_key_buffer := (dept => 100);
          -- Lowest dept for ascending key field.

    start_key_descr:  constant Record_AM.
        key_value_descr := (
            start_key_value'address,
            start_key_value'size / 8);

    stop_key_value:   constant  Employee_Filing_EX.
        dept_key_buffer := (dept => 200);
          -- High end for ascending
          -- key field.

    stop_key_descr:  constant Record_AM.
        key_value_descr := (
            stop_key_value'address,
            stop_key_value'size / 8);
  begin
    Record_AM.Keyed_Ops.Set_key_range(
        opened_dev    => opened_file,
        index         => Employee_Filing_EX.dept_index_name,
        select_range => (
            start_comparison => Record_AM.inclusive,
            start_value      => start_key_descr,
            stop_comparison  => Record_AM.inclusive,
            stop_value       => stop_key_descr));
    loop
      bytes_read := Record_AM.Ops.Read(
          opened_dev => opened_file,
          modifier   => Record_AM.next,
          buffer_VA  => buffer'address,
          length     => buffer'length);

      if current_record_VA.dept = 175 then
        -- CRP does not advance to next record
        -- after the Update_by_key (it advances on
        -- next read).
```

```
      Record_AM.Keyed_Ops.Update_by_key(
          opened_dev => opened_file,
          buffer_VA  => buffer'address,
          length     => buffer'length,
          index      => Employee_Filing_EX.
              dept_index_name);
            -- Employee ID index (hashed).
    end if;
  end loop;
exception
  when Device_Defs.end_of_file =>
    null;
end Read_and_update_by_key;


procedure Read_records_reverse_sequential(
    opened_file:  Device_Defs.opened_device)
      -- An opened device, opened for input on an
      -- employee file.
  --
is
begin
  Record_AM.Ops.Set_position(
      opened_dev => opened_file,
      where      => Record_AM.record_specifier(
          type_of_specifier => Record_AM.last)'(
              type_of_specifier => Record_AM.last));
        -- Positions current record pointer
        -- to last record in file.
  loop
    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        modifier   => Record_AM.prior,
        buffer_VA  => buffer'address,
        length     => buffer'length);

    -- DO ANY NEEDED PROCESSING HERE.

  end loop;


exception
  when Device_Defs.end_of_file =>
    null;
end Read_records_reverse_sequential;


procedure Read_records_sequential(
    opened_file:  Device_Defs.opened_device)
      -- An opened device, opened for input on an
      -- employee file.
  --
is
begin
  Record_AM.Ops.Set_position(
      opened_dev => opened_file,
      where      => Record_AM.record_specifier(
          type_of_specifier => Record_AM.first)'(
              type_of_specifier => Record_AM.first));
  loop
    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length);

    -- DO ANY NEEDED PROCESSING HERE.

  end loop;

exception
  when Device_Defs.end_of_file =>
    null;

end Read_records_sequential;


procedure Read_and_delete_records(
```

```
    opened_file:  Device_Defs.opened_device)
      -- An opened device, opened for input on an
      -- employee file.
is
begin
  Record_AM.Ops.Set_position(
      opened_dev => opened_file,
      where =>      Record_AM.record_specifier(
          type_of_specifier => Record_AM.first)'(
              type_of_specifier => Record_AM.first));
  loop
    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length);
    if current_record_VA.dept = 175 then
      Record_AM.Keyed_Ops.Delete_by_key(
         opened_dev => opened_file,
         index      => Employee_Filing_Ex.
             dept_index_name);
    end if;
  end loop;
exception
  when Device_Defs.end_of_file =>
    null;
end Read_and_delete_records;
procedure Read_and_update_records(
    opened_file:  Device_Defs.opened_device)
      -- An opened device, opened for input on an
      -- employee file.
  --
is
begin
  Record_AM.Ops.Set_position(opened_file,
      where =>      Record_AM.record_specifier(
          type_of_specifier => Record_AM.first)'(
              type_of_specifier => Record_AM.first));
  loop
    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length);
    current_record_VA.salary :=
        pay_raise * current_record_VA.salary;
    Record_AM.Ops.Update(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length);
  end loop;
exception
  when Device_Defs.end_of_file =>
    null;
end Read_and_update_records;
procedure Update_salary_example(
    T2_opened_file:  Device_Defs.opened_device)
      -- An opened device for transaction T1, opened
      -- for input on an employee file.
  --
is
begin
  Transaction_Mgt.Start_transaction;
    -- Started on behalf of transaction T2, the
    -- updater.
```

```
      -- The record must have been positioned to by a
      -- previous read, otherwise a
      -- Record_AM.key_value_descr must be specified.
      -- No key range is necessary. The current record
      -- pointer is not affected.

    current_record_VA.salary :=
        pay_raise * current_record_VA.salary;

    -- Default is the current record.
    Record_AM.Keyed_Ops.Update_by_key(
         opened_dev => T2_opened_file,
         buffer_VA  => buffer'address,
         length     => buffer'length,
         index      => Employee_Filing_EX.
             dept_salary_index_name);
            -- Employee ID index.

  exception
    when Device_Defs.end_of_file =>
      Transaction_Mgt.Commit_transaction;

    when others =>
      Transaction_Mgt.Abort_transaction;

  end Update_salary_example;


end Record_AM_Ex;
```

X-A.4.17 `Simple_editor_cmd_ex` Procedure

```
with Command_Handler,
     Device_Defs,
     Simple_Editor_Ex;

-----------------------------------------------------
--                    SIMPLE EDITOR                    --
-----------------------------------------------------
procedure Simple_editor_cmd_ex
  --
  -- Function:
  --   This procedure implements a simple text
  --   editor for the purpose of demonstrating certain
  --   aspects of the Character Display Access Method.
  --
  -- Command Definition:
  --   The command has the form:
  --
  --     simple_editor_cmd_ex :name=<symbolic_name(1..80)>
  --
  --*D*
  --*D* manage.commands
  --*D*   create.invocation_command
  --*D*
  --*D*   define.argument name \
  --*D*       :type = string
  --*D*       set.lexical_class symbolic_name
  --*D*       set.maximum_length 80
  --*D*       set.mandatory
  --*D*     end
  --*D*   end
  --*D* exit
  --
  -- End of Header

is

  opened_cmd: Device_Defs.opened_device;

begin

  -- Get command arguments:
  --
  opened_cmd := Command_Handler.
      Open_invocation_command_processing;

  Command_Handler.Get_string(
      cmd_odo    => opened_cmd,
      arg_number => 1,
      arg_value  => Simple_Editor_Ex.file_name);

  Command_Handler.Close(opened_cmd);

  -- NOTE: allocation is done here rather than at the
  -- declaration due to the exception
  -- "Object has no representation" being raised
  -- if the Get_object_size is called before the object
  -- is accessed
  Simple_Editor_Ex.edit_buffer :=
     new Simple_Editor_Ex.edit_buffer_object'(
          max_lines => Simple_Editor_Ex.resize_lines,
          num_lines => 0,
          lines     => (others => (others => ASCII.NUL)));

  Simple_Editor_Ex.Read_file;

  Simple_Editor_Ex.Make_window;

  Simple_Editor_Ex.Handle_input;

end Simple_editor_cmd_ex;
```

X-A.4.18 `Simple_Editor_Ex` Package Specification

```
with Incident_Defs,
     System_Defs,
     System,
     Terminal_Defs;

package Simple_Editor_Ex is
  --
  -- Function:
  --   This package implements procedures to support a
  --   simple text editor for the purpose of demonstrating
  --   certain aspects of the Character Display Access Method.
  --
  --   The editor has the following attributes:
  --
  --     1. The file is read into an array of lines of characters.
  --         Each line in 80 characters (screen width)
  --
  --     2.  If the file does not exist it will be created.
  --
  --     3.  The array will expand to any size file.
  --
  --     4.  The array is null-filled before the
  --         file is read in. (Character_Display_AM
  --         will ignore the nulls)
  --
  --     4.  Each line in the file is read into
  --         one row in the array.  Long lines (>80) will be
  --         preserved but they cannot be altered by the editor.
  --
  --     5. The frame buffer is 24 by 80 (screen size).
  --
  --     6. If changes have been made since the last save
  --        it will prompt the user if ok to exit.
  --
  --     7. The bell will ring for illegal commands.
  --
  --   The operations available in the editor are:
  --
  --       * Move forward      (Control F)
  --       * Move backward     (Control B)
  --       * Move up           (Control P)
  --       * Move down         (Control N)
  --       * Page up           (Control U)
  --       * Page down         (Control V)
  --       * Delete forward    (Control D)
  --       * Delete backward   (Control H)
  --       * Insert text
  --       * Save file         (Control W)
  --       * Quit editor       (Control C)
  --
  -- History:
  --   11/??/86, G. Taylor    : Initial version
  --   12/??/87, E. Sassone   : Revised version
  --   12/19/87, G. Taylor    : Added tagged comments
  --   06/15/88, E. Sassone   : working version
  --
  -- Exception Codes:
  --
  new_file_code: constant Incident_Defs.incident_code := (
      module         => 0,
      number         => 1,
      severity       => Incident_Defs.information,
      message_object => System.null_word);

  not_saved_code: constant Incident_Defs.incident_code := (
      module         => 0,
      number         => 2,
      severity       => Incident_Defs.warning,
      message_object => System.null_word);

  no_long_lines_code: constant Incident_Defs.incident_code := (
      module         => 0,
      number         => 3,
      severity       => Incident_Defs.information,
```

```
    message_object => System.null_word);

editor_error_code: constant Incident_Defs.incident_code := (
    module         => 0,
    number         => 4,
    severity       => Incident_Defs.error,
    message_object => System.null_word);
--
-- Exceptions:
--
--*D* manage.messages
--
no_access: exception;
--*D* store :module=0 :number=1 \
--*D*     :msg_name=new_file_code \
--*D*     :short = \
--*D*     "$p1<pathname> is a new file."
--
--*D* store :module=0 :number=2 \
--*D*     :msg_name=not_saved_code \
--*D*     :short = \
--*D*     "Changes have not been saved.  Exit anyway? "
--
--*D* store :module=0 :number=3 \
--*D*     :msg_name=no_long_lines_code \
--*D*     :short = \
--*D*     "Changes to long lines NYI"
--
editor_error: exception;
--*D* store :module=0 :number=4 \
--*D*     :msg_name=editor_error_code \
--*D*     :short = \
--*D*     "Editor_error - please save your file and quit"
--
-- End of Header

------------------------------------------------------
--                      CONSTANTS                   --
------------------------------------------------------
origin:                 constant Terminal_Defs.point_info := (1, 1);
  -- frame buffer origin

first_row:              constant integer   := 1;

first_column:           constant integer   := 1;
last_column:            constant integer   := 80;

frame_rows:             constant integer   := 24;
  -- screen size

preferred_window_rows:  constant integer   := 10;
  -- initial window size

linear_buf_size:        constant := 4_096;
  -- size of read/write buffer

resize_lines:           constant := 100;
  -- number of lines to add for resizing edit buffer
  -- object

------------------------------------------------------
--                      TYPES                       --
------------------------------------------------------
subtype row_delta    is integer range -1 .. 1;
subtype row_range    is positive;
subtype column_range is integer range 1 .. last_column;

-- position in edit_buffer
type cursor_location is
  record
    row:    row_range;
    column: column_range;
  end record;

-- edit buffer
type line            is array (column_range) of character;
type edit_array      is array (integer range <>) of line;
```

```
type edit_buffer_object(
    max_lines: integer) is
  record
    num_lines: integer := 0;
    lines:     edit_array (first_row .. max_lines);
  end record;

type edit_buffer_AD  is access edit_buffer_object;
  pragma access_kind(edit_buffer_AD, AD);

-- for input of command and insertions chars
type char_array      is array (1 .. 120) of character;
type char_array_AD   is access char_array;
  pragma access_kind(char_array_AD, AD);

-----------------------------------------------------
--             VARIABLES                           --
-----------------------------------------------------
file_name:   System_Defs.text(Incident_Defs.txt_length);
edit_buffer: edit_buffer_AD;

-----------------------------------------------------
--             PROCEDURES                          --
-----------------------------------------------------

function Move_page(
    direction: row_delta)
  return boolean;          -- operation successful
  --
  -- Function:
  --  Move up or down by the size of the view
  --


function Move_up
  return boolean;          -- operation successful
  --
  -- Function:
  --   Moves the cursor up one line, but not
  --   beyond the beginning of the file.
  --


function Move_down
  return boolean;          -- operation successful
  --
  -- Function:
  --   Moves the cursor down one line, but not
  --   beyond the end of the file.
  --


function Move_forward
  return boolean;          -- operation successful
  --
  -- Function:
  --   Moves the cursor forward one character
  --   but not beyond the end of the line.
  --


function Move_back
  return boolean;          -- operation successful
  --
  -- Function:
  --   Moves the cursor backward one character, but not
  --   beyond the beginning of the line.
  --


function Delete_forward
  return boolean;          -- operation successful
  --
  -- Function:
  --   Deletes the character at the cursor's current
  --   position. Cursor position in unchanged.
  --
```

```
   function Delete_backward
     return boolean;         -- operation successful
     --
     -- Function:
     --   Deletes the character to the left of the cursor,
     --   but not beyond the beginning of the line.
     --


   function Insert(
       insert_char: character)
     return boolean;         -- operation successful
     --
     -- Function:
     --   Insert printable characters to the left of the
     --   cursor.
     --


   procedure Save_file;
     --
     -- Function:
     --   Writes the file from the edit buffer.
     --


   procedure Quit_editor;
     --
     -- Function:
     --   Exits the editor If changes have been made
     --   since the last save it will ask the user
     --   whether the unsaved changes should be saved or
     --   not.  Returns cursor to old window.
     --


   procedure Read_file;
     --
     -- Function:
     --   Reads the sections of the input file into the
     --   edit buffer.
     --


   procedure Make_window;
     --
     -- Function:
     --   Creates a new window for editing.
     --


   procedure Handle_input;
     --
     -- Function:
     --  Loops waiting for editor keyboard and menu input.
     --

   procedure Key_input(
       key: character);
     --
     -- Function:
     --   Calls the appropriate procedure based on the
     --   key input.
     --

end Simple_Editor_Ex;
```

X-A.4.19 Simple_Editor_Ex Package Body

```
with Byte_Stream_AM,
     Character_Display_AM,
     Device_Defs,
     Directory_Mgt,
     File_Defs,
     Incident_Defs,
     Long_Integer_Defs,
     Message_Services,
     Object_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     Simple_File_Admin,
     System,
     System_Defs,
     Terminal_Defs,
     Text_Mgt,
     Window_Services;

package body Simple_Editor_Ex is

  ------------------------------------------------------
  --             VARIABLES                            --
  ------------------------------------------------------
    -- position of frame buffer in edit_buffer
  frame_begin:       row_range := first_row;
  frame_end:         row_range := frame_rows;

  edit_buf_pos:      cursor_location := (first_row, first_column);

  old_window:        Device_Defs.device;
    -- window editor was invoked from
  edit_window:       Device_Defs.device;
  open_edit_window:  Device_Defs.opened_device;
  saved:             boolean := true;
    -- true if current version has been saved

  ------------------------------------------------------
  --             LAST CHAR IN ROW                     --
  ------------------------------------------------------
  function Last_char_in_row(row: row_range)
    return column_range
    --
    -- Logic:
    --  Starts from the last column of the given row and works
    --  toward the start of the line to detect the first non-null
    --  character.
    --
  is

    column:   column_range := last_column;

  begin

    while edit_buffer.lines(row)(column) = ASCII.NUL
        loop
      if column = first_column then
        EXIT;
      else
        column := column - 1;
      end if;
    end loop;
    return (column);
  end Last_char_in_row;

  ------------------------------------------------------
  --                  MOVE FRAME                      --
  ------------------------------------------------------
  procedure Move_frame(direction: integer)
    --
    -- Logic:
    --   Move frame in edit buffer and rewrite frame buffer.
    --   Reposition cursor appropriately
    --
  is
```

```
  column: column_range := edit_buf_pos.column;
    -- holds cursor position in previous row
begin
  frame_begin := frame_begin + direction;
  frame_end := frame_end + direction;
  edit_buf_pos.row := edit_buf_pos.row + direction;

  Character_Display_AM.Ops.Clear(open_edit_window);

  -- Rewrite frame buffer
  -- NOTE: cursor will be at the end of the frame buffer
  Character_Display_AM.Ops.Write(
      opened_dev => open_edit_window,
      buffer_VA  =>
          edit_buffer.lines(frame_begin)(first_column)'address,
      length     => System.ordinal((last_column * (frame_rows - 1)) +
          Last_char_in_row(frame_end) - 1));

  if direction > 0 then
    -- down:
    -- position at the first column of the last line
    if column > Last_char_in_row(frame_end) then
      column := Last_char_in_row(frame_end);
    end if;
    Character_Display_AM.Ops.Move_cursor_absolute(
        opened_dev => open_edit_window,
        new_pos    => Terminal_Defs.point_info'
            (column, integer(frame_rows)));
  end if;
  if direction < 0 then
    -- up:
    -- after write, cursor will be at last char written
    -- for upward movement we want it at the first char in
    -- the frame buffer
    if column > Last_char_in_row(frame_begin) then
      column := Last_char_in_row(frame_begin);
    end if;
    Character_Display_AM.Ops.Move_cursor_absolute(
        opened_dev => open_edit_window,
        new_pos    => (column, first_row));
  end if;
end Move_frame;

-----------------------------------------------------
--                   MOVE PAGE                     --
-----------------------------------------------------
function Move_page(direction: row_delta)
  return boolean
is
  window_status: Window_Services.window_status :=
      Window_Services.Ops.Get_window_status(
          window      => edit_window,
          pixel_units => false);
  displacement: integer :=
      window_status.window_dimensions.vert * direction;
  cursor_pos:    Terminal_Defs.point_info :=
      Character_Display_AM.Ops.Get_cursor_position(open_edit_window);

begin
  if direction > 0 then
    -- if too close to the bottom move by less than window size
    if frame_end + displacement > edit_buffer.max_lines then
      displacement := edit_buffer.max_lines - frame_end;
    end if;
  end if;
  if direction < 0 then
    -- if too close to the top move by less than window size
    if frame_begin + displacement < first_row then
      displacement := first_row - frame_begin;
    end if;
  end if;

  Move_frame(displacement);
  Character_Display_AM.Ops.Move_cursor_absolute(
      opened_dev => open_edit_window,
      new_pos    => cursor_pos);
  edit_buf_pos.row := frame_begin + (cursor_pos.vert - 1);
```

```
  if displacement = 0 then
    return false;
  else
    return true;
  end if;
end Move_page;


-----------------------------------------------------
--                  MOVE CURSOR                    --
-----------------------------------------------------
procedure Move_cursor(direction: row_delta)
is

  -- used for current cursor position
  cursor_pos: Terminal_Defs.point_info :=
      Character_Display_AM.Ops.Get_cursor_position(open_edit_window);
  -- last column of row where cursor will be
  last_col: column_range := Last_char_in_row(edit_buf_pos.row +
      direction);

begin

  edit_buf_pos.row := edit_buf_pos.row + direction;

  if cursor_pos.horiz <= last_col then
  -- Move cursor in frame buffer straight up or down
  Character_Display_AM.Ops.Move_cursor_relative(
      opened_dev => open_edit_window,
      delta_col  => 0,
      delta_row  => direction);
  else
  -- Move cursor to end of line
    Character_Display_AM.Ops.Move_cursor_absolute(
        opened_dev => open_edit_window,
        new_pos    => (last_col, edit_buf_pos.row));
    edit_buf_pos.column := last_col;
  end if;
end Move_cursor;


-----------------------------------------------------
--                 MOVE UP                         --
-----------------------------------------------------
function Move_up
  return boolean

is
  success: boolean := true;
begin
  if edit_buf_pos.row <= first_row then
    success := false;
  elsif edit_buf_pos.row <= frame_begin then
    Move_frame(-1);
  else
    Move_cursor(-1);
  end if;
  return success;
end Move_up;


-----------------------------------------------------
--                  MOVE DOWN                      --
-----------------------------------------------------
function Move_down
  return boolean

is
  success: boolean := true;
begin

  if edit_buf_pos.row >= edit_buffer.num_lines then
    success := false;
  elsif edit_buf_pos.row >= frame_end then
    Move_frame(+1);
  else
    Move_cursor(+1);
```

```
    end if;
    return success;
  end Move_down;


  ------------------------------------------------------
  --                MOVE FORWARD                      --
  ------------------------------------------------------
  function Move_forward
    return boolean
    --
    -- Logic:
    --   If cursor is at end of row then move cursor to
    --   first column of next row; else move cursor
    --   forward one column.  If cursor is at the end of
    --   of the buffer return false.
    --
  is
    current_pos: Terminal_Defs.point_info;
    success: boolean := true;
  begin

    if edit_buf_pos.column = Last_char_in_row(edit_buf_pos.row) then
      if edit_buf_pos.row = edit_buffer.num_lines then
        success := false;   -- at the end of buffer
      else
        -- Move cursor to next row in frame and
        --  frame buffer
        success := Move_down;
        if not success then return success; end if;
        -- Move cursor to beginning of row in frame
        -- and frame buffer
        current_pos := Character_Display_AM.Ops.
            Get_cursor_position(open_edit_window);
        current_pos.horiz := first_column;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => open_edit_window,
            new_pos    => current_pos);
        edit_buf_pos.column := first_column;
      end if;
    else
      -- move cursor to next column
      edit_buf_pos.column := edit_buf_pos.column + 1;
      Character_Display_AM.Ops.Move_cursor_relative(
          opened_dev => open_edit_window,
          delta_col  => 1,
          delta_row  => 0);
    end if;
    return success;
  end Move_forward;


  -----------------------------------------------------
  --             MOVE BACK                           --
  -----------------------------------------------------
  function Move_back
    return boolean
    --
    -- Logic:
    --   If cursor is at beginning of row then move cursor
    --   to last column of previous row; else move cursor
    --   back one column. If cursor is at the beginning of
    --   the file then return false.

  is
    current_pos: Terminal_Defs.point_info;
    success:     boolean := true;
  begin

    if edit_buf_pos.column = first_column then
      if edit_buf_pos.row = first_row then
        Character_Display_AM.Ops.Ring_bell(
            open_edit_window);
      success := false;
      else
        -- Move cursor to previous row in frame and
        -- frame buffer
```

```
      success := Move_up;
      if not success then return success; end if;
      -- Move cursor to end of row
      edit_buf_pos.column := last_char_in_row(edit_buf_pos.row);
      current_pos := Character_Display_AM.Ops.
          Get_cursor_position(open_edit_window);
      current_pos.horiz := edit_buf_pos.column;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => open_edit_window,
          new_pos    => current_pos);
    end if;
  else
    -- move cursor to previous column
    edit_buf_pos.column := edit_buf_pos.column - 1;
    Character_Display_AM.Ops.Move_cursor_relative(
        opened_dev => open_edit_window,
        delta_col  => -1,
        delta_row  => 0);
  end if;
  return success;
end Move_back;


----------------------------------------------------
--               DELETE FORWARD                   --
----------------------------------------------------
function Delete_forward
  return boolean
  --
  -- Logic:
  --   Procedure will not delete characters from long
  --   lines.  It then determines if the the character
  --   to be deleted is a line feed or not.  If not it
  --   simple deletes the character and shifts
  --   characters beyond it one position to the left.
  --   If the character is a line feed it determines if
  --   the line is empty or not.  If so if deletes the
  --   line.  If not it joins the current line with the
  --   next line.  In both cases lines beyond the
  --   current line are shifted up by one row.
  --
is
  -- place holders for line joins
  cursor_pos: Terminal_Defs.point_info :=
      Character_Display_AM.Ops.Get_cursor_position( open_edit_window);
  edit_pos: cursor_location := edit_buf_pos;

begin

  -- no deletes on long lines
  if Last_char_in_row(edit_buf_pos.row) = last_column then
    Message_Services.Write_msg(no_long_lines_code);
    return false;
  end if;
  if edit_buf_pos.column = Last_char_in_row(edit_buf_pos.row) then
    if edit_buf_pos.row = edit_buffer.num_lines then
      return false;
    end if;
  end if;

  -- not a line feed
  if edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column)
      /= ASCII.LF then
    -- Delete the character from the frame.
    if edit_buf_pos.column = last_column then
        edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column)
            := ASCII.NUL;
    else
      for col in edit_buf_pos.column..last_column - 1 loop
        edit_buffer.lines(edit_buf_pos.row)(col) :=
            edit_buffer.lines(edit_buf_pos.row)(col + 1);
      end loop;
    end if;

  edit_buffer.lines(edit_buf_pos.row)(last_column) := ASCII.NUL;

  -- Delete the character from the window.
```

```
  Character_Display_AM.Ops.Delete_char(
      opened_dev => open_edit_window);

  -- line feed
  else
    -- not the last line
    if edit_buf_pos.row < edit_buffer.num_lines then
      -- empty line  delete
      if edit_buf_pos.column = first_column then
        -- shift rows down by one
        for row in edit_buf_pos.row .. edit_buffer.num_lines - 1
        loop
          edit_buffer.lines(row) := edit_buffer.lines(row + 1);
        end loop;
        edit_buffer.lines(edit_buffer.num_lines) :=
            (others => ASCII.NUL);
        edit_buffer.num_lines := edit_buffer.num_lines - 1;
        Character_Display_AM.Ops.Delete_line(open_edit_window);
      -- join current line and next line
      else
        -- don't join if line wiil be too long
        if Last_char_in_row(edit_buf_pos.row) +
          Last_char_in_row(edit_buf_pos.row + 1) >= last_column then
          return false;
        end if;
        for col in first_column .. Last_char_in_row(
          edit_buf_pos.row + 1)
        loop
          edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column) :=
              edit_buffer.lines(edit_buf_pos.row + 1)(col);
          edit_buf_pos.column := edit_buf_pos.column + 1;
          EXIT when edit_buf_pos.column = last_column;
        end loop;
        edit_buf_pos.row := edit_buf_pos.row + 1;
        -- shift rows down by one
        for row in edit_buf_pos.row .. edit_buffer.num_lines - 1
        loop
          edit_buffer.lines(row) := edit_buffer.lines(row + 1);
        end loop;
        edit_buffer.lines(edit_buffer.num_lines) :=
            (others => ASCII.NUL);
        edit_buffer.num_lines := edit_buffer.num_lines - 1;
        Move_frame(0);    -- redraw
        edit_buf_pos := edit_pos;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => open_edit_window,
            new_pos    => cursor_pos);
      end if;
    -- last line
    else
      edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column) :=
          ASCII.NUL;
    end if;
  end if;
  return true;
end Delete_forward;


-----------------------------------------------------
--              DELETE BACKWARD                    --
-----------------------------------------------------
function Delete_backward
  return boolean
  --
  -- Logic:
  --   Very similar to Delete_forward except the cursor
  --   is move back before the delete is performed.
  --
is
  success: boolean := true;
  res:     boolean;
begin

  if Move_back then   -- back up cursor
    success := Delete_forward;   -- Delete the character.
    -- leave cursor pos unchanged if unsuccessful
    if not success then res := Move_forward; end if;
```

```
  else
    success := false;
  end if;
  return success;
end Delete_backward;


---------------------------------------------------
--                     INSERT                    --
---------------------------------------------------
function Insert(insert_char: character)
  return boolean
  --
  -- Logic:
  --   Shifts the string of characters beginning at the
  --   cursor's location one character position to the
  --   right.  It then inserts a  printable ASCII character
  --   to the left of the cursor.  If a line is already
  --   80 characters the insert is refused.  Line feeds
  --   are inserted by first moving all the rows beyond the
  --   current row down by one. If there are characters
  --   on the current line beyond the insert point they
  --   are copied to the new line.  If not just a line-
  --   feed in put into the new line.  If the file grows
  --   beyond the current max_line size it is expanded by
  --   resize lines.
  --
is

  use System;  -- for adding System.ordinals

  max_lines: integer;
  For max_lines USE AT edit_buffer.max_lines'address;

  edit_buffer_untyped:  System.untyped_word;
  FOR edit_buffer_untyped USE AT edit_buffer'address;

  -- place holders for line splits
  cursor_pos: Terminal_Defs.point_info :=
     Character_Display_AM.Ops.Get_cursor_position(open_edit_window);
  edit_pos: cursor_location := edit_buf_pos;
  column: column_range := first_column;

  success: boolean := true;

begin

  -- inserts on long lines NYI
  if Last_char_in_row(edit_buf_pos.row) = last_column then
    Message_Services.Write_msg(no_long_lines_code);
    return false;
  end if;

  -- If the current column is the last column in the
  -- view, insert the new character in the frame;
  -- else shift trailing characters one column to
  -- the right and insert the new character.
  --
  if insert_char /= ASCII.LF then
    if edit_buf_pos.column = last_column then
      edit_buffer.lines(edit_buf_pos.row)
          (edit_buf_pos.column) := insert_char;
    else
      -- right shift characters to the right of insert position
      for index in reverse edit_buf_pos.column + 1 .. last_column
      loop
        edit_buffer.lines(edit_buf_pos.row)(index) :=
            edit_buffer.lines(edit_buf_pos.row)(index - 1);
      end loop;

      edit_buffer.lines(edit_buf_pos.row)
          (edit_buf_pos.column) := insert_char;
      edit_buf_pos.column := edit_buf_pos.column + 1;
    end if;

    -- Insert the character in the frame buffer
    -- (Frame buffer cursor is moved automatically)
```

```
    Character_Display_AM.Ops.Insert_char(
        opened_dev => open_edit_window,
        buffer_VA  => insert_char'address,
        num_char   => 1);

  -- return
  else
    -- shift buffer lines beyond current row down by one
      if edit_buffer.num_lines + 1 >= edit_buffer.max_lines then
        -- add resize_lines lines to current edit buffer size
        Object_Mgt.Resize(
            obj  => edit_buffer_untyped,
            size => (Object_Mgt.Get_object_size(
                edit_buffer_untyped) +
                ordinal((resize_lines * last_column) / 4)));
        max_lines := edit_buffer.num_lines + resize_lines;
        edit_buffer.lines(edit_buffer.num_lines + 1 ..
            edit_buffer.max_lines) := (others => (others => ASCII.NUL));
      end if;

    -- move row down one
    for row in reverse edit_buf_pos.row + 1 .. edit_buffer.num_lines
    loop
      edit_buffer.lines(row + 1) := edit_buffer.lines(row);
    end loop;
    -- blank fill line below current line
    edit_buffer.lines(edit_buf_pos.row + 1) := (others => ASCII.NUL);
    edit_buffer.num_lines := edit_buffer.num_lines + 1;

    -- add return to end of line
    if edit_buf_pos.column = Last_char_in_row(edit_buf_pos.row) then
      success := Move_down;
      -- first char of new line in LF
      edit_buffer.lines(edit_buf_pos.row)(first_column) := ASCII.LF;
      edit_buf_pos.column := first_column;
      Character_Display_AM.Ops.Insert_line(open_edit_window);
    -- insert return in the middle of the line (split line)
    else
      -- copy characters past point of insert to the next line
      for col in edit_buf_pos.column .. Last_char_in_row(edit_buf_pos.row)
      loop
        edit_buffer.lines(edit_buf_pos.row + 1)(column) :=
            edit_buffer.lines(edit_buf_pos.row)(col);
        -- clear line past point of insert
        edit_buffer.lines(edit_buf_pos.row)(col) := ASCII.NUL;
        edit_buf_pos.column := edit_buf_pos.column + 1;
        column := column + 1;
      end loop;
      edit_buffer.lines(edit_pos.row)(edit_pos.column) := ASCII.LF;
      Move_frame(0);                      -- redraw
      edit_buf_pos.row := edit_buf_pos.row + 1;
      edit_buf_pos.column := first_column;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => open_edit_window,
          new_pos    => Terminal_Defs.point_info'(
              first_column, cursor_pos.vert + 1));
    end if;
  end if;
  return success;
end Insert;


------------------------------------------------------
--                SAVE FILE                          --
------------------------------------------------------
procedure Save_file
  --
  -- Logic:
  --   Writes the file in linear_buf_size amounts copied
  --   from the edit_buffer which is an array of lines
  --   to the linear buffer.  It checks for backslashes
  --   in the last column and rejoins long lines.
  --   Before writing the new file, it must be truncated
  --   and the pointer moved back to zero.
  --
is
```

```
  opened_file: Device_Defs.opened_device;
  file_ptr:    Long_Integer_Defs.long_integer;
  linear_buffer: array (1 .. linear_buf_size) of
    character := (others => ASCII.NUL);
  index:         integer := 1;

begin

  opened_file :=  Byte_Stream_AM.Open_by_name(
      name         => file_name,
      input_output => Device_Defs.output,
      allow        => Device_Defs.nothing);

  -- delete data in original file
  Byte_Stream_AM.Ops.Truncate(
      opened_dev => opened_file,
      new_length => Long_Integer_Defs.zero);

  file_ptr := Byte_Stream_AM.Ops.Set_position(
      opened_dev => opened_file,
      pos        => Long_Integer_Defs.zero,
      mode       => Byte_Stream_AM.from_begin);

  for row in 1 .. edit_buffer.num_lines
  loop
    -- write each line to linear buffer until LF
    for col in first_column .. last_column
    loop
      -- write out linear buffer when full;
      if index > linear_buf_size then
        Byte_Stream_AM.Ops.Write(
            opened_dev => opened_file,
            buffer_VA  => linear_buffer'address,
            length     => System.ordinal(linear_buffer'size / 8));
        linear_buffer := (others => ASCII.NUL);
        index := 1;
      end if;
      -- reproduce long lines
      if col < last_column or
          edit_buffer.lines(row)(last_column) /= '\' then
        linear_buffer(index) := edit_buffer.lines(row)(col);
        index := index + 1;
        EXIT when edit_buffer.lines(row)(col) = ASCII.LF;
     end if;
    end loop;
  end loop;

  Byte_Stream_AM.Ops.Write(
      opened_dev => opened_file,
      buffer_VA  => linear_buffer'address,
      length     => System.ordinal(index));

  Byte_Stream_AM.Ops.Close(opened_file);

exception

  when Directory_Mgt.no_access =>
      Message_Services.Write_msg(
          msg_id => new_file_code,
          param1 => Incident_Defs.message_parameter'(
              typ     => Incident_Defs.txt,
              len     => file_name.length,
              txt_val => file_name));

end Save_file;


-----------------------------------------------------
--                QUIT EDITOR                      --
-----------------------------------------------------
procedure Quit_editor

is
  quit:           exception;
begin

  Window_Services.Ops.Transfer_input_focus(
```

```
      source_window => edit_window,
      target_window => old_window);

  if not saved then
    if not Message_Services.Acknowledge_Msg(not_saved_code) then
      Window_Services.Ops.Transfer_input_focus(
          source_window => old_window,
          target_window => edit_window);
      return;
   end if;
  end if;

  Character_Display_AM.Ops.Close(open_edit_window);
  Window_Services.Ops.Destroy_window(edit_window);
  RAISE quit;

exception

  when quit =>
      RAISE;

end Quit_editor;

-----------------------------------------------------
--                CLOSE INPUT                      --
-----------------------------------------------------
procedure Close_input
  -- NYI (for menus)
is
begin
  null;
end Close_input;

-----------------------------------------------------
--                READ FILE                        --
-----------------------------------------------------
procedure Read_file
  --
  -- Logic:
  --   Reads the input file into a linear buffer.
  --   That is read one line feed to a row into
  --   the edit buffer.  The edit buffer is expanded
  --   by resize lines increments as needed.
  --   A backslash is place in the last column for
  --   lines over 80 characters long.
  --
is

  use System;   -- for adding System.ordinals

  characters_read:      System.ordinal;
  opened_file:          Device_Defs.opened_device;
  linear_buffer:        array (1 .. linear_buf_size) of character;
  col, row:             integer := 1;
  file:                 File_Defs.file_AD;

  max_lines:            integer;
  For max_lines USE AT edit_buffer.max_lines'address;

  edit_buffer_untyped:  System.untyped_word;
  FOR edit_buffer_untyped USE AT edit_buffer'address;

begin

  opened_file :=  Byte_Stream_AM.Open_by_name(
      name         => file_name,
      input_output => Device_Defs.input);

  loop

    -- read by linear_buf_size blocks
    characters_read := Byte_Stream_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => linear_buffer'address,
        length     => System.ordinal(linear_buffer'size / 8));

    for index in 1 .. integer(characters_read)
```

```
      loop
        if row > max_lines then
          -- add resize_lines lines to current edit buffer size
          Object_Mgt.Resize(
              obj  => edit_buffer_untyped,
              size => (Object_Mgt.Get_object_size(
                  edit_buffer_untyped) +
                  ordinal((resize_lines * last_column) / 4)));
          max_lines := edit_buffer.num_lines + resize_lines;
          -- initialize expanded area
          edit_buffer.lines(edit_buffer.num_lines + 1 ..
              edit_buffer.max_lines) :=
                  (others => (others => ASCII.NUL));
        end if;

        if linear_buffer(index) = ASCII.LF then
          edit_buffer.lines(row)(col) := linear_buffer(index);
          edit_buffer.num_lines := edit_buffer.num_lines + 1;
          col := 1;
          row := row + 1;
        else
          if col < last_column then
            edit_buffer.lines(row)(col) := linear_buffer(index);
            col := col + 1;
          else   -- long line
            edit_buffer.lines(row)(last_column) := '\';
            edit_buffer.num_lines := edit_buffer.num_lines + 1;
            col := 1;
            row := row + 1;
            edit_buffer.lines(row)(col) := linear_buffer(index);
          end if;
        end if;
      end loop;
    end loop;

    Byte_Stream_AM.Ops.Close(opened_file);

  exception

    -- make a new file
    when Directory_Mgt.no_access =>
        Message_Services.Write_msg(
            msg_id => new_file_code,
            param1 => Incident_Defs.message_parameter'(
                typ     => Incident_Defs.txt,
                len     => file_name.length,
                txt_val => file_name));
        file := Simple_File_Admin.Create_file(file_name);
        RETURN;

    -- successful completion
    when Device_Defs.end_of_file =>
        Byte_Stream_AM.Ops.Close(opened_file);

  end Read_file;

  -----------------------------------------------------
  --                    MAKE WINDOW                  --
  -----------------------------------------------------
  procedure Make_window
  is

    underlying_terminal:        Device_Defs.device;
    new_window_info:    Window_Services.window_style_info;
    window_attributes:  Terminal_Defs.window_attr :=
        Terminal_Defs.default_window_attr;

  begin

    -- Create new window from old opened window.
    old_window := Character_Display_AM.Ops.
        Get_device_object(Process_Mgt.Get_process_globals_entry(
            Process_Mgt_Types.standard_input));
    underlying_terminal := Window_Services.Ops.
        Get_terminal(old_window);
    edit_window := Window_Services.Ops.Create_window(
        terminal             => underlying_terminal,
```

```
        pixel_units         => false,
        fb_size             => Terminal_Defs.point_info'(
            last_column, frame_rows),
        desired_window_size => Terminal_Defs.point_info'(
            last_column, preferred_window_rows),
        window_pos          => origin,
        view_pos            => origin);
    -- Set window's input and output attributes
    -- change from default:
    window_attributes.enable_signal := false;  -- for ^C ^B
    window_attributes.line_editing := false;   -- for ^H
    window_attributes.echo := false;
    -- NOTE: track_cursor NYI (use user agent to change view)
    window_attributes.track_cursor := true;
    Window_Services.Ops.Set_window_attr(
        window    => edit_window,
        attr      => window_attributes,
        attr_mask => (others => true));
    -- Set Title and Info lines
    Text_Mgt.Set(new_window_info.title, file_name);
    Window_Services.Ops.Set_window_style(
        window     => edit_window,
        new_info   => new_window_info,
        style_list => (others => true));

    -- Open the edit window
    open_edit_window := Character_Display_AM.Ops.Open(
        device          => edit_window,
        input_output    => Device_Defs.inout,
        exclusive       => true);

    -- Clear window on terminal screen.
    Character_Display_AM.Ops.Clear(open_edit_window);

    -- Write from edit buffer to frame buffer.
    -- NOTE: There cannot be more line_feeds in the length
    -- of characters written than there are rows in
    -- the frame buffer, otherwise some of the first
    -- characters will be overwritten in the frame buffer
    -- The last line is written up to the line feed to
    -- avoid having a blank line at bottom of the window
    Character_Display_AM.Ops.Write(
        opened_dev => open_edit_window,
        buffer_VA  => edit_buffer.lines'address,
        length     => System.ordinal((last_column * (frame_rows - 1))
            + (Last_char_in_row(frame_end) - 1)));


    -- Home the cursor (1,1 position).
    Character_Display_AM.Ops.Move_cursor_absolute(
        opened_dev => open_edit_window,
        new_pos    => origin);

    Window_Services.Ops.Transfer_input_focus(
        source_window => old_window,
        target_window => edit_window);

  end Make_window;


  -----------------------------------------------------
  --                    HANDLE INPUT                 --
  -----------------------------------------------------
  procedure Handle_input

  is

    event_num:        System.ordinal;
    event_type:       Terminal_Defs.input_enum;
    char_buffer_AD:   char_array_AD := new char_array'(others => ' ');

  begin

    -- Enter the basic read and process loop
    loop
      -- Read the next input event
      -- default input mask is keyboard
```

```
    Character_Display_AM.Ops.Read(
        opened_dev => open_edit_window,
        buffer_VA  => char_buffer_AD.all'address,
        max_events => 1,
        max_bytes  => 0,
        block      => true,
        type_read  => event_type,
        num_read   => event_num);
      case event_type is
        when Terminal_Defs.keyboard =>
           -- ...
            key_input(char_buffer_AD(1));
        when Terminal_Defs.menu_item_picked =>
           -- ...
            key_input(char_buffer_AD(1));
        when others =>
            null;
      end case;
    end loop;
end Handle_input;

-----------------------------------------------------
--                    IS PRINTABLE                 --
-----------------------------------------------------
function Is_printable(c: character)
  return boolean
--
-- Logic:
--  Checks if character entered in printable
--
is
begin

  if c >= ' ' or c = ASCII.LF then return true;
  else return false;
  end if;

end Is_printable;
  pragma inline(Is_printable);
-----------------------------------------------------
--                    KEY INPUT                    --
-----------------------------------------------------
procedure Key_input(key: character)

is
  result:     boolean := true;
  cursor_pos: Terminal_Defs.point_info;
begin

    -- Process the event
  case key is
    when ASCII.ACK =>
        result := Move_forward;         -- Control F
    when ASCII.STX =>
        result := Move_back;            -- Control B
    when ASCII.DLE =>
        result := Move_up;              -- Control P
    when ASCII.SO  =>
        result := Move_down;            -- Control N
    when ASCII.NAK  =>
        result := Move_Page(-1);        -- Control U
    when ASCII.SYN  =>
        result := Move_page(+1);        -- Control V
    when ASCII.EOT =>
        result := Delete_forward;       -- Control D
        saved := false;
    when ASCII.BS  =>
        result := Delete_backward;      -- Control H
        saved := false;
    when ASCII.ETB  =>
        Save_file;                      -- Control W
        saved := true;
    when ASCII.ETX  =>
        Quit_editor;                    -- Control C
    when others    =>
       --Insert text.
       if Is_printable(key) then
```

```
           result := Insert(key);
           saved := false;
         else Character_Display_AM.Ops.Ring_bell(open_edit_window);
         end if;
    end case;
    if not result then
      Character_Display_AM.Ops.Ring_bell(open_edit_window);
    end if;
    -- cursor check
    cursor_pos := Character_Display_AM.Ops.Get_cursor_position(
        open_edit_window);
    if edit_buf_pos.row /= frame_begin + (cursor_pos.vert - 1) or
        edit_buf_pos.column /= cursor_pos.horiz then
      RAISE editor_error;
    end if;

  exception

    when editor_error =>
        Message_Services.Write_msg(editor_error_code);
        return;

  end Key_input;

end Simple_Editor_Ex;
```

X-A.4.20 Stream_file_ex Procedure

```
with Directory_Mgt,
     File_Defs,
     Passive_Store_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     Simple_File_Admin,
     System_Defs,
     Text_Mgt;

procedure Stream_file_ex is
  --
  -- Function:
  --   Provide example calls for stream files.

  filename:  System_Defs.text(60);
  file1:     File_Defs.file_AD;
  file2:     File_Defs.file_AD;
  file3:     File_Defs.file_AD;
begin
  Text_Mgt.Set(filename, "my_file_1");
  file1 := Simple_File_Admin.Create_file(filename);
    -- Creates a stream file in the current
    -- directory.

    -- Code to write something into the file
    -- could go here.

  Text_Mgt.Set(filename, "my_file_2");
  file2 := Simple_File_Admin.Create_file(filename);
  Simple_File_Admin.Copy_file(source_file => file1,
                              target_file => file2);
    -- Creates a second file in the current directory,
    -- and then copies the contents of the first file
    -- to the second.

  Simple_File_Admin.Empty_file(file1);
    -- Empties the first file.

  Text_Mgt.Set(filename, "my_file_2");
  Directory_Mgt.Delete(filename);
    -- The second file's pathname is deleted.  The
    -- second file is destroyed when the last
    -- reference to it goes away.

  file2 := Simple_file_Admin.Create_unnamed_file(
      Passive_Store_Mgt.Home_volume_set(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.current_dir)));
    -- Creates a temporary file in the current
    -- directory using the current directory's
    -- volume set.

  Text_Mgt.Set(filename, "my_local_name");
  Simple_File_Admin.Save_unnamed_file(
      name => filename,
      file => file2);
    -- Names and saves the temporary file so that it
    -- can be used in future jobs.

  file3 := Simple_file_Admin.Create_unnamed_file(
      Passive_Store_Mgt.Home_volume_set(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.current_dir)));
    -- Creates another temporary file in the current
    -- directory.

  Simple_File_Admin.Destroy_file(file3);
  -- Destroys the temporary file before its job
  -- terminates.  If it is not destroyed or saved,
  -- it goes away when the job terminates.

end Stream_file_ex;
```

X-A.5 Human Interface Services

X-A.5.1 Inventory_main Procedure

```
with Character_Display_AM,
     Device_Defs,
     Incident_Defs,
     Inventory_Files,
     Inventory_Menus,
     Inventory_Messages,
     Inventory_Windows,
     Message_Services,
     System,
     Terminal_Defs;

  --
  -- Function:
  --   Main (top-level) procedure for Inventory
  --   Example Program.
  --
  --   The procedure "Inventory_main" is called from
  --   CLEX.  "Inventory_main" performs the top-level
  --   processing for the Inventory Example Program:
  --   program initialization, main processing loop,
  --   and termination.
  --
  -- History:
  --   05-20-87, William A. Rohm:  Written.
  --   10-27-87, WAR:              Revised.
  --
  -- End of Header

procedure Inventory_main
  --
  -- Logic:
  --   1. Define incident codes.
  --   2. Open windows and files.
  --   3. Install and enable menu group, enable menu
  --      selection
  --   4. Process each menu selection until Exit
  --   5. Close files and windows.
is

  -- Incident codes for messages:
  --
  module:  constant := 1;
    -- Message module index number.

  -- *M*   set.language  :language = English
  -- *M*   create.variable  module  :value = 1


  welcome_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 0,
          severity       => Incident_Defs.information);

  -- *M*   store  :module = $module  :number = 0\
  -- *M*          :msg_name = welcome \
  -- *M*          :short = "Welcome to the Inventory
  -- *M*                    Example Program."


  terminated_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 1,
          severity       => Incident_Defs.information);

  -- *M*   store  :module = $module  :number = 1\
  -- *M*          :msg_name = terminated \
  -- *M*          :short = "Inventory Example Program
  -- *M*                    terminated."
```

```
  -- Variables:
  --
    menu_select:  Terminal_Defs.menu_selection;
       -- Menu selection record for receiving user
       -- input from "Character_Display_AM.Ops.Read".
       --
       -- Contains user's menu group, menu, and item
       -- selection numbers.

    event_type:  Terminal_Defs.input_enum;
      -- Type of user input event (returned from
      -- "Character_Display_AM.Ops.Read").

    event_num:   System.ordinal;
       -- Number of user input events (returned from
       -- "Character_Display_AM.Ops.Read").


-- Inventory_main procedure:
--
  begin

    -- Open both main and message windows:
    --
    Inventory_Windows.Open_program_windows;


    -- Display "Welcome" message:
    --
    Message_Services.Write_msg(
        msg_id => welcome_code);


    -- Open files:
    --
    Inventory_Files.Open_parts_file;

    Inventory_Files.Open_log_file;


    -- Retrieve and install menu group:
    --
    Inventory_Menus.Set_up_menu_group;


    -- Set input event type mask for menu item selection
    -- only:
    --
    Character_Display_AM.Ops.Set_input_type_mask(
        opened_dev => Inventory_Windows.main_window,
        new_mask   => Terminal_Defs.input_type_mask'(
            Terminal_Defs.menu_item_picked => true,
            others                         => false));


    -- Main processing loop:
    --
    loop

      -- Wait for and read next input event (must have
      -- been a menu selection):
      --
      Character_Display_AM.Ops.Read(
          opened_dev => Inventory_Windows.main_window,
          buffer_VA  => menu_select'address,
          max_events => 1,
          max_bytes  => 0,
          block      => true,   -- Wait . . .
          type_read  => event_type,
          num_read   => event_num);


      -- Act on menu selection:
      --
      case menu_select.menu is
```

```
      when Inventory_Menus.inquiry_menu_ID =>
          Inventory_Menus.Process_inquiry_menu(
              selection  => menu_select.item);

      when Inventory_Menus.posting_menu_ID =>
          Inventory_Menus.Process_posting_menu(
              selection => menu_select.item);

      when Inventory_Menus.update_menu_ID =>
          Inventory_Menus.Process_update_menu(
              selection => menu_select.item);

      when Inventory_Menus.report_menu_ID =>
          Inventory_Menus.Process_report_menu(
              selection => menu_select.item);

      when Inventory_Menus.housekeeping_menu_ID =>
          Inventory_Menus.Process_housekeeping_menu(
              selection => menu_select.item);

      when Inventory_Menus.exit_menu_ID =>
          EXIT;

      when others =>
          null;

    end case;         -- "case menu_select.menu is"

  end loop;


  -- Close files:
  --
  Inventory_Files.Close_parts_file;

  Inventory_Files.Close_log_file;


  -- Write "terminated" message:
  --
  Message_Services.Write_msg(
      msg_id => terminated_code);


  -- Close both program windows.  When the main
  -- window is closed, the menu group is deallocated:
  --
  Inventory_Windows.Close_program_windows;

end Inventory_main;
```

X-A.5.2 `Inventory_Files` Package Specification

```
with Device_Defs,
     Incident_Defs,
     Inventory_Messages,
     System,
     System_Defs,
     Timing_Conversions;

package Inventory_Files is
  --
  -- Function:
  --   Contains all operations related to
  --   Inventory Program files.
  --
  --   This package contains the necessary calls
  --   to open and close the two inventory files
  --   (parts file and log file), and calls to
  --   read and write records in the parts file,
  --   and to write records to the log file.
  --
  -- History:
  --   05-20-87, William A. Rohm:  Written.
  --   11-02-87, WAR:              Revised.
  --
  -- End of Header

  -- Incident codes for messages:
  --
  module:  constant := 3;
    -- Message module index.

  --*M*   set.language  :language=english
  --*M*   create.variable  module  :value = 3

  no_modify_rights_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 1,
          severity       => Incident_Defs.error);


  --*M*   store  :module = $module  \
  --*M*          :number = 1 \
  --*M*          :msg_name = no_mod_rights \
  --*M*          :short = "No modify rights for
  --*M*             parts file '$p1<parts file
  --*M*             name>'."



  no_parts_file_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 2,
          severity       => Incident_Defs.error);

  --*M*   store  :module = $module  \
  --*M*          :number = 2 \
  --*M*          :msg_name = no_parts_file \
  --*M*          :short = "Parts file '$p1<parts
  --*M*            file name>' does not
  --*M*            exist."


  no_log_file_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 3,
          severity       => Incident_Defs.error);
```

```
--*M*   store  :module = $module \
--*M*          :number = 3 \
--*M*          :msg_name = no_log_file \
--*M*          :short = "Log file '$p1<log
--*M*            file name>' does not
--*M*            exist."


index_inconsistent_code:  constant
    Incident_Defs.incident_code := (
        message_object =>
            Inventory_Messages.message_object,
        module         => module,
        number         => 4,
        severity       => Incident_Defs.error);

 --*M*   store  :module = $module \
 --*M*          :number = 4 \
 --*M*          :msg_name = \
 --*M*              index_inconsistent \
 --*M*          :short = "Parts file
 --*M*            '$p1<parts file name>' index
 --*M*            is inconsistent and must be
 --*M*            redone.  Select the
 --*M*            Housekeeping Menu's item
 --*M*            'Index Parts File'."


not_on_file_code:  constant
    Incident_Defs.incident_code := (
        message_object =>
            Inventory_Messages.message_object,
        module         => module,
        number         => 5,
        severity       => Incident_Defs.error);

--*M*   store  :module = $module \
--*M*          :number = 5 \
--*M*          :msg_name = not_on_file \
--*M*          :short = "There is no parts
--*M*            record for part ID '$p1<part
--*M*            ID (index value)>' does not
--*M*            exist."

not_on_file:  exception;
  pragma exception_value(not_on_file,
                         not_on_file_code);
  -- Raised by "Read_parts_record" and
  -- "Rewrite_parts_record".


invalid_part_ID_code:  constant
    Incident_Defs.incident_code := (
        message_object =>
            Inventory_Messages.message_object,
        module         => module,
        number         => 6,
        severity       => Incident_Defs.error);

--*M*   store  :module = $module \
--*M*          :number = 6 \
--*M*          :msg_name = invalid_part_ID \
--*M*          :short = "An invalid part ID,
--*M*             '$p1<part ID (index
--*M*             value)>', was entered."

invalid_part_ID:  exception;
  pragma exception_value(invalid_part_ID,
                         invalid_part_ID_code);
  -- Raised by "Read_parts_record",
  -- "Write_parts_record", and
  -- "Rewrite_parts_record".


already_on_file_code:  constant
    Incident_Defs.incident_code := (
```

```
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 7,
          severity       => Incident_Defs.error);

--*M*     store  :module = $module \
--*M*            :number = 7 \
--*M*            :msg_name = already_on_file \
--*M*            :short = "Parts record for part
--*M*               ID '$p1<part ID (index
--*M*               value)>' already exists.
--*M*               Either choose a new part ID,
--*M*               or update the current part's
--*M*               record."

already_on_file:  exception;
  pragma exception_value(already_on_file,
                         already_on_file_code);
  -- Raised by "Read_parts_record" and
  -- "Write_parts_record".


-- Constants:
--
parts_file_str:  constant string :=
    "/example/inventory/parts_file";
  -- String constant for parts file's
  -- pathname.

parts_file_pathname:  System_Defs.text(
    Incident_Defs.txt_length) := (
        Incident_Defs.txt_length,
        parts_file_str'length,
        parts_file_str);
  -- Text constant from parts file's pathname
  -- string.
part_ID_index_str:  constant string :=
    "part_ID_index";
  -- String constant for parts file's
  -- index's name.

part_ID_index_name:  System_Defs.text(
    part_ID_index_str'length) := (
        part_ID_index_str'length,
        part_ID_index_str'length,
        part_ID_index_str);
  -- Text constant from parts file's index's
  -- name string.

log_file_str:  constant string :=
    "/example/inventory/log_file";
  -- String constant for log file's
  -- pathname.

log_file_pathname:  System_Defs.text(
    Incident_Defs.txt_length) := (
        Incident_Defs.txt_length,
        log_file_str'length,
        log_file_str);
  -- Text constant from log file's pathname
  -- string.


-- Variables:
--
parts_file:  Device_Defs.opened_device;
  -- AD to inventory parts file.

log_file:    Device_Defs.opened_device;
  -- AD to inventory log file.


--------------------------------------------------
--      Inventory Parts File Record Definition
--------------------------------------------------
```

```
  -- Constants:
  --
  part_ID_length:      constant integer :=  Incident_Defs.txt_length;
  desc_length:         constant integer := 30;
  unit_length:         constant integer :=  4;
  loc_length:          constant integer := 12;
  status_length:       constant integer :=  7;
  max_suppliers:       constant integer :=  3;
  supplier_ID_length:  constant integer := 10;

  qty_digits:          constant integer :=  7;


  -- Types:
  --
  subtype part_ID_type is System_Defs.text(
      part_ID_length);

  subtype supplier_ID_type is System_Defs.text(
      supplier_ID_length);

  subtype location_type is System_Defs.text(
      loc_length);

--type qty_type is digits qty_digits;
  subtype qty_type is System.ordinal
            range 0..9_999_999;

--type cost_type is delta 0.01
--                  range 0.0 .. 99_999_999.99;
  subtype cost_type is float
      range 0.0..99_999_999.99;

  type supplier_array_type is
    array (1..max_suppliers) of supplier_ID_type;
    -- Array of supplier IDs.

  type parts_record_type is
    -- Record declaration for parts file
    -- records.
    record
      part_ID:            part_ID_type;
        -- Part identification code (ID).
      desc:               System_Defs.text(
          desc_length);
        -- Description of part.
      unit:               System_Defs.text(
          unit_length);
        -- Unit of measure.
      location:           location_type;
        -- Warehouse location of part.
      qty_on_hand:        qty_type;
      reorder_point:      qty_type;
      reorder_qty:        qty_type;
      suppliers:          supplier_array_type;
        -- Array of suppliers for this part.
      usage_this_month:   qty_type;
      usage_last_month:   qty_type;
      usage_last_year:    qty_type;
      avg_unit_cost:      cost_type;
      last_unit_cost:     cost_type;
      date_first_act:
          Timing_Conversions.numeric_time;
        -- Date and time of first activity with
        -- this part (entered into parts file).
      date_last_act:
          Timing_Conversions.numeric_time;
        -- Date and time of last activity with
        -- this part.
      status:             System_Defs.text(
          status_length);
        -- Status of this part ("on order", "on
        -- hold", "obsolete", ...).
  end record;


-------------------------------------------------
```

```
--      Inventory Log File Record Definition
-------------------------------------------------

  -- Constants:
  --
  doc_length:       constant integer := 12;
  job_length:       constant integer := 32;

  -- Types:
  --
  type action_type is (
      create,
        -- Create new parts record
      update,
        -- Update parts record
      delete,
        -- Delete parts record
      receipt,
      issue,
      returns,
      spoilage,
      journal);


  type log_record_type is
    -- Record declaration for log file records.
    record
      part_ID:       part_ID_type;
        -- Part ID used in current action.
      action:        action_type;
        -- Action performed with this part ID.
      time:
          Timing_Conversions.numeric_time;
        -- Date and time of action.
      doc_number:    System_Defs.text(
          doc_length);
        -- Supplier's document number.
      qty:           qty_type;
        -- Taken from
        -- "parts_file_record.qty_on_hand".
      job_ID:        System_Defs.text(
          job_length);
        -- ID of job which called Inventory
        -- Example Program to perform action.
      supplier_ID:  supplier_ID_type;
        -- ID of supplier for this part and
        -- action.
   end record;


-- Parts file procedures:
--   Open / Read / Write / Rewrite / Close file
--

  procedure Open_parts_file;
    --
    -- Function:
    --   Opens inventory parts file.


  procedure Read_parts_record(
      part_ID:            part_ID_type;
        -- Part ID of record to be read.
      msg_on_error:       boolean := false;
        -- Optional parameter specifying whether
        -- a message is displayed when an
        -- exception is raised.  Default is no
        -- message.
      parts_record:  out parts_record_type);
        -- Variable that receives parts record.
    --
    -- Function:
    --   Reads a record from the inventory parts
    --   file.
    --
    -- Exceptions:
    --   not_on_file - "part_ID" does not index
```

```
    --                 an existing parts record.
    --   invalid_part_ID - "part_ID" contains an
    --                     invalid value.


  procedure Write_parts_record(
      parts_record:  parts_record_type);
        -- Record to be written.
    --
    -- Function:
    --   Writes a record to the inventory parts
    --   file.
    --
    -- Exceptions:
    --   already_on_file - "part_ID" indexes
    --                     an existing parts record.
    --   invalid_part_ID - "part_ID" contains an
    --                     invalid value.


  procedure Rewrite_parts_record(
      parts_record:  parts_record_type);
        -- Record to be rewritten.
    --
    -- Function:
    --   Rewrites a record in the inventory
    --   parts file.
    --
    -- Exceptions:
    --   not_on_file - "part_ID" does not index
    --                 an existing parts record.
    --   invalid_part_ID - "part_ID" contains an
    --                     invalid value.


  procedure Delete_parts_record(
      part_ID:  part_ID_type);
        -- ID of record to be deleted.
    --
    -- Function:
    --   Deletes a record in the inventory parts file.
    --
    -- Exceptions:
    --   not_on_file - "part_ID" does not index
    --                 an existing parts record.
    --   invalid_part_ID - "part_ID" contains an
    --                     invalid value.


  procedure Close_parts_file;
    --
    -- Function:
    --   Closes inventory parts file.


-- Log file procedures:
--   Open / Write / Close log file
--

  procedure Open_log_file;
    --
    -- Function:
    --   Opens inventory log file.


  procedure Write_log_record(
      parts_record:  parts_record_type;
        -- Affected parts record.
      action:        action_type);
        -- Action taken with parts record.
    --
    -- Function:
    --   Creates and writes a record to the inventory
    --   log file.


  procedure Close_log_file;
```

```
    --
    -- Function:
    --    Closes inventory log file.

end Inventory_Files;
```

X-A.5.3 Inventory_Files Package Body

```
with Access_Mgt,
     Device_Defs,
     Directory_Mgt,
     Incident_Defs,
     Inventory_Windows,
     Message_Services,
     Message_Stack_Mgt,
     Object_Mgt,
     Record_AM,
     System,
     System_Defs,
     System_Exceptions,
     Timed_Requests_Mgt,
     Timing_Conversions,
     Unchecked_conversion;


package body Inventory_Files is
  --
  -- Function:
  --   Contains all operations related to Inventory
  --   Program files.
  --
  -- History:
  --   05-20-87, William A. Rohm:  Written.
  --   10-27-87, WAR:              Revised.
  --
  -- End of Header


-- Generic function:
--
function Device_from_untyped_word is new
    Unchecked_conversion(
        source => System.untyped_word,
        target => Device_Defs.device);


-- Parts file procedures:
--  Open / Read / Write / Rewrite / Close parts file

  procedure Open_parts_file
  is
    parts_file_AD:  System.untyped_word;

  begin

    -- Retrieve parts file, if possible:
    --
    parts_file_AD := Directory_Mgt.Retrieve(
        name => parts_file_pathname);

    -- Check for access (modify) rights for parts file:
    --
    if not Access_Mgt.Permits(
        AD     => parts_file_AD,
        rights => Object_Mgt.modify_rights)
    then
      Message_Services.Write_msg(
          msg_id => no_modify_rights_code,
          param1 => Incident_Defs.message_parameter(
              typ => Incident_Defs.txt,
              len => parts_file_pathname.length)'(
                  typ     => Incident_Defs.txt,
                  len     => parts_file_pathname.length,
                  txt_val => parts_file_pathname));
    end if;
    -- Open parts file:
    --
    parts_file := Record_AM.Ops.Open(
        dev          => Device_from_untyped_word(
            parts_file_AD),
        input_output => Device_Defs.inout,
        allow        => Device_Defs.readers);
```

```

  exception
    -- Exceptions from "Directory_Mgt.Retrieve",
    -- "Record_AM.Ops.Open":
    --
    when others =>
      Message_Services.Write_msg(
          msg_id => no_parts_file_code,
          param1 => Incident_Defs.message_parameter(
              typ => Incident_Defs.txt,
              len => parts_file_pathname.length)'(
                  typ     => Incident_Defs.txt,
                  len     => parts_file_pathname.length,
                  txt_val => parts_file_pathname));

  end Open_parts_file;


  procedure Read_parts_record(
      part_ID:            part_ID_type;
      msg_on_error:       boolean := false;
      parts_record:   out parts_record_type)
  is
    bytes_read:  System.ordinal;

  use System;    -- To import "/=" for
                 -- "System.ordinal", and division for
                 -- "'size/8" constructions

  begin

    -- Read given record, if any:
    --
    bytes_read := Record_AM.Keyed_Ops.Read_by_key(
        opened_dev => parts_file,
        buffer_VA  => parts_record'address,
        length     => parts_record'size/8,
        index      => part_ID_index_name,
        key_buffer => Record_AM.key_value_descr'(
            buffer_VA => part_ID'address,
            length    => part_ID'size/8));

--    if bytes_read /= parts_record'size/8 then
--      -- msg "Couldn't get record"
--    end if;

  exception

    when Record_AM.invalid_record_address =>

      if msg_on_error then
        Message_Services.Write_msg(
            msg_id => not_on_file_code,
            param1 => Incident_Defs.message_parameter(
                typ => Incident_Defs.txt,
                len => part_ID.length)'(
                    typ     => Incident_Defs.txt,
                    len     => part_ID.length,
                    txt_val => part_ID));
        Message_Stack_Mgt.Push_msg_1_param(
            not_on_file_code);
      end if;

      RAISE not_on_file;

    when Record_AM.key_value_incomplete =>

      if msg_on_error then
        Message_Services.Write_msg(
            msg_id => invalid_part_ID_code,
            param1 => Incident_Defs.message_parameter(
                typ => Incident_Defs.txt,
                len => part_ID.length)'(
                    typ     => Incident_Defs.txt,
                    len     => part_ID.length,
                    txt_val => part_ID));
        Message_Stack_Mgt.Clear_messages;
```

```
      Message_Stack_Mgt.Push_msg_1_param(
          message_id => invalid_part_ID_code,
          param1     => Incident_Defs.message_parameter(
              typ => Incident_Defs.txt,
              len => part_ID.length)'(
                  typ     => Incident_Defs.txt,
                  len     => part_ID.length,
                  txt_val => part_ID));
    end if;

    RAISE invalid_part_ID;


  when Record_AM.index_inconsistent =>

    Message_Services.Write_msg(
        msg_id => index_inconsistent_code,
        param1 => Incident_Defs.message_parameter(
            typ => Incident_Defs.txt,
            len => parts_file_pathname.length)'(
                typ     => Incident_Defs.txt,
                len     => parts_file_pathname.length,
                txt_val => parts_file_pathname));

  when others =>
    RAISE;

end Read_parts_record;


procedure Write_parts_record(
    parts_record:  parts_record_type)
is

use System;   -- For "'size/8" constructions

begin

  -- Write (insert in index key sequence) new record
  -- into parts file:
  --
  Record_AM.Ops.Insert(
      opened_dev => parts_file,
      buffer_VA  => parts_record'address,
      length     => parts_record'size/8);

exception
  when Record_AM.invalid_duplicate =>
    RAISE already_on_file;

  when Record_AM.invalid_record_address |
       Record_AM.key_value_incomplete =>
    RAISE invalid_part_ID;

  when Record_AM.index_inconsistent =>
    Message_Services.Write_msg(
        msg_id => index_inconsistent_code,
        param1 => Incident_Defs.message_parameter(
            typ => Incident_Defs.txt,
            len => parts_file_pathname.length)'(
                typ     => Incident_Defs.txt,
                len     => parts_file_pathname.length,
                txt_val => parts_file_pathname));

  when others =>
    RAISE;

end Write_parts_record;


procedure Rewrite_parts_record(
    parts_record:  parts_record_type)
is

use System;   -- for "'size/8" constructions

begin
```

```


  -- Rewrite (update) parts record:
  --
  Record_AM.Keyed_Ops.Update_by_key(
      opened_dev => parts_file,
      buffer_VA  => parts_record'address,
      length     => parts_record'size/8,
      index      => part_ID_index_name);

exception

  when Record_AM.invalid_record_address =>
    Message_Services.Write_msg(
        msg_id => not_on_file_code,
        param1 => Incident_Defs.message_parameter(
            typ => Incident_Defs.txt,
            len => part_ID_index_str.length)'(
                typ     => Incident_Defs.txt,
                len     => part_ID_index_str.length,
                txt_val => part_ID_index_name));
    RAISE not_on_file;

  when Record_AM.key_value_incomplete =>
    RAISE invalid_part_ID;

  when Record_AM.index_inconsistent =>
    Message_Services.Write_msg(
        msg_id => index_inconsistent_code,
        param1 => Incident_Defs.message_parameter(
            typ => Incident_Defs.txt,
            len => parts_file_pathname.length)'(
                typ     => Incident_Defs.txt,
                len     => parts_file_pathname.length,
                txt_val => parts_file_pathname));

  when others =>
    RAISE;

end Rewrite_parts_record;


procedure Delete_parts_record(
    part_ID:  part_ID_type)
is

use System;   -- for "'size/8" constructions

begin

  -- Delete parts record:
  --
  Record_AM.Keyed_Ops.Delete_by_key(
      opened_dev => parts_file,
      index      => part_ID_index_name,
      key_buffer => Record_AM.key_value_descr'(
          buffer_VA => part_ID'address,
          length    => part_ID'size/8));

exception

  when Record_AM.invalid_record_address =>
    RAISE not_on_file;

  when Record_AM.key_value_incomplete =>
    RAISE invalid_part_ID;

  when Record_AM.index_inconsistent =>
    Message_Services.Write_msg(
        msg_id => index_inconsistent_code,
        param1 => Incident_Defs.message_parameter(
            typ => Incident_Defs.txt,
            len => parts_file_pathname.length)'(
                typ     => Incident_Defs.txt,
                len     => parts_file_pathname.length,
                txt_val => parts_file_pathname));

```

```
    when others =>
      RAISE;

  end Delete_parts_record;


  procedure Close_parts_file
  is

  begin

    if Record_AM.Ops.Is_open(parts_file) then
      Record_AM.Ops.Close(
          opened_dev => parts_file);
    end if;

  end Close_parts_file;


-- Log file procedures:
--   Open / Write / Close log file

  procedure Open_log_file
  is
    log_file_AD:  System.untyped_word;

  begin

    -- Retrieve log file, if possible:
    --
    log_file_AD := Directory_Mgt.Retrieve(
        log_file_pathname);

    -- Check for access (modify) rights for log file:
    --
    if not Access_Mgt.Permits(
        AD     => log_file_AD,
        rights => Object_Mgt.modify_rights)
    then
      Message_Services.Write_msg(
          msg_id => no_modify_rights_code,
          param1 => Incident_Defs.message_parameter(
              typ => Incident_Defs.txt,
              len => log_file_pathname.length)'(
                  typ     => Incident_Defs.txt,
                  len     => log_file_pathname.length,
                  txt_val => log_file_pathname));
    end if;

    -- Open log file:
    --
    log_file := Record_AM.Ops.Open(
        dev          => Device_from_untyped_word(
            log_file_AD),
        input_output => Device_Defs.inout,
        allow        => Device_Defs.nothing,
        block        => false);

    exception
    -- Exceptions from "Directory_Mgt.Retrieve",
    -- "Record_AM.Ops.Open":
    --
      when others =>
        Message_Services.Write_msg(
            msg_id => no_log_file_code,
            param1 => Incident_Defs.message_parameter(
                typ => Incident_Defs.txt,
                len => log_file_pathname.length)'(
                    typ     => Incident_Defs.txt,
                    len     => log_file_pathname.length,
                    txt_val => log_file_pathname));
  end Open_log_file;


  procedure Write_log_record(
      parts_record:  parts_record_type;
      action:        action_type)
```

```
  is
    log_record:  log_record_type;

  use System;   -- for "'size/8" constructions

  begin

    log_record.part_ID := parts_record.part_ID;

    log_record.action := action;

    log_record.time := Timing_Conversions.
        Convert_stu_to_numeric_time(
           stu => Timed_Requests_Mgt.Get_time);

    log_record.doc_number := System_Defs.text(doc_length)'
        (doc_length, 0, (others => ' '));

    log_record.qty := parts_record.qty_on_hand;

    log_record.job_ID := System_Defs.text(job_length)'
        (job_length, 0, (others => ' '));

    log_record.supplier_ID :=
        parts_record.suppliers(1);

    Record_AM.Ops.Set_position(
        opened_dev => log_file,
        where      => Record_AM.record_specifier(
            type_of_specifier => Record_AM.last)'(
                type_of_specifier => Record_AM.last));

    Record_AM.Ops.Insert(
        opened_dev => log_file,
        buffer_VA  => log_record'address,
        length     => log_record'size/8);

  end Write_log_record;



  procedure Close_log_file
  is

  begin

    if Record_AM.Ops.Is_open(log_file) then
      Record_AM.Ops.Close(
          opened_dev => log_file);
    end if;

  end Close_log_file;


end Inventory_Files;
```

X-A.5.4 `Inventory_Forms` Package Specification

```
with Device_Defs,
     Form_Defs,
     Incident_Defs,
     Inventory_Files,
     Inventory_Messages,
     System,
     System_Defs,
     Terminal_Defs;

package Inventory_Forms is
  --
  -- Function:
  --   Contains subprograms to display and process
  --   Inventory Program forms.
  --
  --   Includes form handling routines
  --   ("Process_*x*_form"), a form processing routine
  --   ("Validate_cost"), and two key-catcher routines
  --   ("Go_to_inquiry" and "Add_supplier_ID").
  --
  -- History:
  --   07-06-87, William A. Rohm:  Written.
  --   11-02-87, WAR:              Revised.
  --
  -- End of Header

  -- Incident codes for messages:
  --
  module:  constant := 5;
    -- Message module index.

  --*M*   set.language  :language = English
  --*M*   create.variable  module  :value = 5

  invalid_output_device_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 0,
          severity       => Incident_Defs.error);

  --*M*   store  :module = 5  :number = 0\
  --*M*          :msg_name = invalid_output_dev\
  --*M*          :short = "Entered output device
  --*M*                    pathname '$p1<pathname>'
  --*M*                    does not exist, or does
  --*M*                    not support the record
  --*M*                    access method."


  unit_cost_error_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 1,
          severity       => Incident_Defs.warning);

  --*M*   store  :module = 5  :number = 1\
  --*M*          :msg_name = cost_error\
  --*M*          :short = "Entered part's unit
  --*M*             cost is not within
  --*M*             $p1<allowed variation
  --*M*             percentage>% of the average
  --*M*             unit cost.  Please re-enter
  --*M*             $p2<total/unit> cost, or the
  --*M*             number of units."


  -- Constants:
  --
  inquiry_form_str:  constant string :=
      "/examples/inventory/forms/inquiry";
```

```
  -- String constant for inquiry form's
  -- pathname.

inquiry_form_pathname:  System_Defs.text(
    inquiry_form_str'length) := (
        inquiry_form_str'length,
        inquiry_form_str'length,
        inquiry_form_str);
  -- Text constant from inquiry form's
  -- pathname string.

receipts_form_str:  constant string :=
    "/examples/inventory/forms/receipts";
  -- String constant for receipts form's
  -- pathname.

receipts_form_pathname:  System_Defs.text(
    receipts_form_str'length) := (
        receipts_form_str'length,
        receipts_form_str'length,
        receipts_form_str);
  -- Text constant from receipts form's
  -- pathname string.

update_form_str:  constant string :=
    "/examples/inventory/forms/update";
  -- String constant for update form's
  -- pathname.

update_form_pathname:  System_Defs.text(
    update_form_str'length) := (
        update_form_str'length,
        update_form_str'length,
        update_form_str);
  -- Text constant from update form's
  -- pathname string.

report_form_str:  constant string :=
    "/examples/inventory/forms/report";
  -- String constant for report form's
  -- pathname.

report_form_pathname:  System_Defs.text(
    report_form_str'length) := (
        report_form_str'length,
        report_form_str'length,
        report_form_str);
  -- Text constant from report form's
  -- pathname string.


----------------------------------------------
-- Field and subform names for forms:
--
part_ID_field:      System_Defs.text( 7) := (
    7,7,  "part_ID");
desc_field:         System_Defs.text(11) := (
    11,11,"description");
unit_field:         System_Defs.text( 4) := (
    4,4,  "unit");
loc_field:          System_Defs.text( 8) := (
    8,8,  "location");
qty_field:          System_Defs.text(11) := (
    11,11,"qty_on_hand");
reorder_pt_field:   System_Defs.text(13) := (
    13,13,"reorder_point");
reorder_qty_field:  System_Defs.text(11) := (
    11,11,"reorder_qty");
suppliers_field:    System_Defs.text( 9) := (
    9,9,  "suppliers");
usage_tmo_field:    System_Defs.text(16) := (
    16,16,"usage_this_month");
usage_lmo_field:    System_Defs.text(16) := (
    16,16,"usage_last_month");
usage_lyr_field:    System_Defs.text(15) := (
    15,15,"usage_last_year");
avg_cost_field:     System_Defs.text(13) := (
```

```
        13,13,"avg_unit_cost");
    last_cost_field:     System_Defs.text(14) := (
        14,14,"last_unit_cost");
    date_first_field:    System_Defs.text(14) := (
        14,14,"date_first_act");
    date_last_field:     System_Defs.text(13) := (
        13,13,"date_last_act");
    status_field:        System_Defs.text( 6) := (
        6,6,  "status");

    inq_suppl_ref_field:  System_Defs.text(19) := (
        19,19,"supplier_ref_number");
    inq_date_field:       System_Defs.text( 4) := (
        4,4,  "date");
    inq_time_field:       System_Defs.text( 4) := (
        4,4,  "time");

    rpt_type_field:      System_Defs.text(11) := (
        11,11,"report_type");
    rpt_opt_field:       System_Defs.text(14) := (
        14,14,"report_options");
    rpt_dev_field:       System_Defs.text(20) := (
        20,20,"report_output_device");


    -- Group and subform names for forms:
    --
    inq_part_ID_only:  System_Defs.text(16) := (
        16,16,"inq_part_ID_only");
    inq_all:           System_Defs.text(15) := (
        15,15,"inq_display_all");

    update_add:        System_Defs.text(10) := (
        10,10,"update_add");
    update_change:     System_Defs.text(13) := (
        13,13,"update_change");
    update_delete:     System_Defs.text(13) := (
        13,13,"update_delete");

    --
    ---------------------------------------------



    -- Types:
    --
    subtype percentage is System.short_ordinal
        range 0..99;

    type percentage_range_type is
      -- Type for containing percentage range.
      record
        percent_less:  percentage;
          -- Maximum percent of change less than
          -- reference value.
        percent_more:  percentage;
          -- Maximum percent of change more than
          -- reference value.
      end record;


    procedure Process_inquiry_form;
      -- Function:
      --   Processes inquiry form:  displays form in
      --   main window, gets valid information
      --   ("part_ID"), then reads Parts Master File and
      --   displays parts record.


    procedure Process_receipts_form;
      -- Function:
      --   Processes receipts form:  displays form in
      --   main window, gets valid information
      --   ("part_ID", "supplier", "quantity", etc),
      --   reads Parts Master File to validate, updates
      --   parts record, then writes log file record.
```

```
procedure Process_update_form(
    selection:  Terminal_Defs.menu_item_ID);
      -- Selection made in the "Maintenance" menu;
      -- either *Add*, *Change*, or *Delete*.
  -- Function:
  --   Processes update form:  displays form in main
  --   window, gets valid information ("part_ID"),
  --   reads Parts Master File and displays parts
  --   record, then updates or deletes part record.


procedure Process_report_form(
    report_by_part:      boolean;
      -- True if the report is to be "by part",
      -- false if the report is "by location".
    report_out_dev:  out System_Defs.text);
      -- Variable that receives output device
      -- pathname where report is to be sent.
  --
  -- Function:
  --   Processes report form:  displays form in main
  --   window, gets report output device.


-- Form processing & key catcher routines:
--

procedure Validate_cost(
    old_parts_record:
        Inventory_Files.parts_record_type;
      -- Parts record from file.
    qty_received:
        Inventory_Files.qty_type;
      -- Entered quantity received.
    total_cost:        in out
        Inventory_Files.cost_type;
      -- Entered or calculated total cost.
    unit_cost:         in out
        Inventory_Files.cost_type;
      -- Entered or calculated unit cost.
    total:                     boolean := true;
      -- Whether to calculate the "total_cost" from
      -- the "unit_cost", or vice versa.
      --
      -- If true (default), the "total_cost" is
      -- calculated from the given "unit_cost" times
      -- the given "qty_received".  If false, the
      -- "unit_cost" is calculated by dividing the
      -- given "total_cost" by the given
      -- "qty_received".
    percentage_range:
        percentage_range_type := (5, 5);
      -- Maximum low and high percentage
      -- difference between parts record's
      -- "avg_unit_cost" (also required of
      -- "last_unit_cost") and the entered or
      -- calculated "unit_cost" parameter.
    valid:                   out boolean);
      -- Whether the entered or calculated unit cost
      -- is within the given "percentage_range" of
      -- cost on file.
--
-- Function:
--   Processing routine called from the Receipts
--   form to validate unit cost and to calculate and
--   return either total cost or unit cost.


procedure Go_to_inquiry;
--
-- Function:
--   Key catcher called from the "Receipts" or
--   "Change" form when the user presses the
--   "<Go-to-Inquiry>" key.  Calls
--   "Process_inquiry_form".
--
--   When this procedure (key-catcher) is activated,
```

```
    --   the enclosing form has been suspended.  When
    --   this procedure returns, the enclosing form
    --   continues.


    procedure Add_supplier_ID(
        opened_form:  Form_Defs.opened_form_AD);
          -- Opened form to which another "supplier_ID"
          -- field will be added.
    --
    -- Function:
    --   Key catcher called from the "Add" form when the
    --   user presses the "<next>" key.  Adds another
    --   "supplier_ID" field to current form, up to a
    --   total of three.

end Inventory_Forms;
```

X-A.5.5 Inventory_Forms Package Body

```
with Data_Definition_Mgt,
     Device_Defs,
     Directory_Mgt,
     Form_Defs,
     Form_Handler,
     Inventory_Files,
     Inventory_Menus,
     Inventory_Windows,
     Message_Services,
     Record_AM,
     System,
     System_Defs,
     Terminal_Defs,
     Text_Mgt,
     Timed_Requests_Mgt,
     Timing_Conversions,
     Unchecked_Conversion,
     Window_Services;


package body Inventory_Forms is


  function Get_form(
      form_pathname:  System_Defs.text)
    return Form_Defs.opened_form_AD
  is
    --
    -- Logic:
    --   Gets requested form from directory, opens
    --   form.

    -- Generic function:
    --
    function DDef_from_untyped is new
        Unchecked_conversion(
             source => System.untyped_word,
             target => Data_Definition_Mgt.DDef_AD);

    opened_form:  Form_Defs.opened_form_AD;
      -- Returned opened form's AD.

  begin
    opened_form := Form_Handler.Open_form(
        DDef => DDef_from_untyped(
                Directory_Mgt.Retrieve(
                name => form_pathname)));

    return opened_form;

  end Get_form;


  procedure Process_inquiry_form
    --
    -- Logic:
    --   1. Display form in main window
    --   2. Get valid information ("part_ID")
    --   3. Read Parts Master File and display parts
    --      record

  is

    opened_form:  Form_Defs.opened_form_AD;
    form_status:  Form_Defs.status_t;

    opened_record_form:  Device_Defs.opened_device;
      -- For record access to "opened_form".

    part_ID:       Inventory_Files.part_ID_type;
    parts_record:  Inventory_Files.parts_record_type;

    length:        System.ordinal;
    empty:         boolean;
```

```
  error:          boolean;

  first_time:     boolean := true;

use Form_Defs;    -- import "/=" for type
                  -- "Form_Defs.status_t"

use System;       -- for "'size/8" arithmetic

begin

  opened_form := Get_form(inquiry_form_pathname);

  -- Open form's DDef for record access:
  --
  opened_record_form := Record_AM.Open_by_name(
      name         => inquiry_form_pathname,
      input_output => Device_Defs.inout);

  -- Set up first rank (group) in "inquiry form"
  -- pile:
  --
  Form_Handler.Create_group_instances(
      opened_form_a       => opened_form,
      group               => inq_part_ID_only,
      number_of_instances => 1);


  -- Read part ID, display, ask for another:
  --
  loop

    -- Get first part ID:
    --
    form_status := Form_Handler.Get(
        opened_form_a   => opened_form,
        opened_window_a => Inventory_Windows.
                             main_window);

    if form_status /= Form_Defs.finished then

      -- some kind of error processing
      null;

    else
      Form_Handler.Fetch_value(
          opened_form_a        => opened_form,
          element              => part_ID_field,
          subunit              => System_Defs.null_text,
            -- added subunit; value correct?
          value_buffer_VA      => part_ID'address,
          value_length         => part_ID'size/8,
          value_t              =>
              Data_Definition_Mgt.t_string,
          element_value_length => length,
          empty                => empty);

      if empty then -- user entered null part ID:
                    -- exit loop; return to menu
        EXIT;
      end if;

      -- Read parts file, handle exceptions:
      --
      begin

        Inventory_Files.Read_parts_record(
          part_ID      => part_ID,
          msg_on_error => true,
          parts_record => parts_record);

        if first_time then
          -- set up other rank

          first_time := false;

          -- Remove first group (rank):
```

```
            --
            Form_Handler.Remove_group_instances(
                opened_form_a       => opened_form,
                group               => inq_part_ID_only,
                number_of_instances => 1);

            -- Add second group (rank):
            --
            Form_Handler.Create_group_instances(
                opened_form_a       => opened_form,
                group               => inq_all,
                number_of_instances => 1);

          end if;  -- If "first_time" through

          Record_AM.Ops.Update(
              opened_dev => opened_record_form,
              buffer_VA  => parts_record'address,
              length     => parts_record'size/8);

        exception
          when Inventory_Files.not_on_file =>
            null;       -- "Record not found" message
                        -- has been displayed; go
                        -- through loop again

          when Inventory_Files.invalid_part_ID =>
            null;       -- "Invalid part ID entered"
                        -- message has been displayed;
                        -- go through loop again
        end;

      end if;  -- if form status = finished

    end loop;  -- read part_ID, display loop

    Form_Handler.Clear(
        opened_form_a => opened_form);

    Form_Handler.Close_form(
        opened_form_a => opened_form);

    -- Close record access to form:
    --
    Record_AM.Ops.Close(
        opened_dev => opened_record_form);

  end Process_inquiry_form;



  procedure Process_receipts_form
    --
    -- Logic:
    --   1. Display form in main window
    --   2. Get receipt information ("part_ID",
    --      "supplier", etc)
    --   3. Read Parts Master File to validate
    --   4. If valid, update parts record, then write
    --      log file record.

  is

    opened_form:  Form_Defs.opened_form_AD;
    form_status:  Form_Defs.status_t;

    part_ID:       Inventory_Files.part_ID_type;
    parts_record:  Inventory_Files.parts_record_type;

    length:        System.ordinal;
    empty, error:  boolean;

    now:           Timing_Conversions.numeric_time;

  use Form_Defs;  -- import "/=" for type
                  -- "Form_Defs.status_t"

```

```
use System;     -- for "'size/8" arithmetic

begin

  opened_form := Get_form(receipts_form_pathname);

  loop

    form_status := Form_Handler.Get(
        opened_form_a   => opened_form,
        opened_window_a => Inventory_Windows.
                           main_window);

    if form_status /= Form_Defs.finished then

      -- Some kind of error processing
      null;

    else


      Form_Handler.Fetch_value(
          opened_form_a        => opened_form,
          element              => part_ID_field,
          subunit              => System_Defs.null_text,
            -- added subunit; value correct?
          value_buffer_VA      => part_ID'address,
          value_length         => part_ID'size/8,
          value_t              =>
              Data_Definition_Mgt.t_string,
          element_value_length => length,
          empty                => empty);

      if empty then
        -- null part_ID; return to menu

        EXIT;
      end if;

      begin

        Inventory_Files.Read_parts_record(
            part_ID      => part_ID,
            msg_on_error => true,
            parts_record => parts_record);


        Form_Handler.Store_value(
            opened_form_a   => opened_form,
            element         => desc_field,
            subunit         => System_Defs.null_text,
              -- added subunit; value correct?
            value_buffer_VA =>
                parts_record.desc'address,
            value_length    =>
                parts_record.desc'size/8,
            value_t         =>
                Data_Definition_Mgt.t_string);

        Form_Handler.Store_value(
            opened_form_a   => opened_form,
            element         => unit_field,
            subunit         => System_Defs.null_text,
              -- added subunit; value correct?
            value_buffer_VA =>
                parts_record.unit'address,
            value_length    =>
                parts_record.unit'size/8,
            value_t         =>
                Data_Definition_Mgt.t_string);

        now := Timing_Conversions.
            Convert_stu_to_numeric_time(
                stu => Timed_Requests_Mgt.Get_time);

        Form_Handler.Store_value(
            opened_form_a   => opened_form,
```

```
            element         => inq_date_field,
            subunit         => System_Defs.null_text,
              -- added subunit; value correct?
            value_buffer_VA => now'address,
            value_length    => now'size/8,
            value_t         =>
                Data_Definition_Mgt.t_date);

        Form_Handler.Store_value(
            opened_form_a   => opened_form,
            element         => inq_time_field,
            subunit         => System_Defs.null_text,
              -- added subunit; value correct?
            value_buffer_VA => now'address,
            value_length    => now'size/8,
            value_t         =>
                Data_Definition_Mgt.t_date);

        exception

          when Inventory_Files.not_on_file =>
            null;     -- "Record not found" message
                      -- has been displayed; go
                      -- through loop again

          when Inventory_Files.invalid_part_ID =>
            null;     -- "Invalid part ID entered"
                      -- message has been displayed;
                      -- go through loop again

        end; -- Read parts record block

    end if;           -- if form status = finished

  end loop;

  Form_Handler.Clear(
      opened_form_a => opened_form);

  Form_Handler.Close_form(
      opened_form_a => opened_form);

end Process_receipts_form;


procedure Process_update_form(
    selection:  Terminal_Defs.menu_item_ID)
  --
  -- Logic:
  --   1. Get update form and create appropriate
  --      subform
  --   2. Get "part_ID"
  --   3. Read Parts Master File and display parts
  --      record
  --   4. Add, change, or delete part record
  --   5. Write appropriate log record

is

  opened_form:  Form_Defs.opened_form_AD;
    -- AD to opened "update" form.

  form_status:  Form_Defs.status_t;

  part_ID:           Inventory_Files.part_ID_type;
  parts_record:
      Inventory_Files.parts_record_type;
  new_parts_record:
      Inventory_Files.parts_record_type;
  log_record:        Inventory_Files.log_record_type;

  opened_record_form:  Device_Defs.opened_device;
    -- For record access to "opened_form".

  length:       System.ordinal;
    -- Length of a returned record, in bytes.
  empty:        boolean;
```

```
      -- Whether the entered "part_ID" field was
      -- empty.

    new_part:     boolean;
      -- True if this is a new part ID (add only!).

  use Form_Defs;          -- to import "/=" for
                          -- Form_Defs.status_t

  use System;             -- for "'size/8" arithmetic

  begin

    -- Open "update" form:
    --
    opened_form := Get_form(
        update_form_pathname);

    -- Create appropriate group instance
    -- (add, change, delete):
    --
    case selection is

      when Inventory_Menus.update_add_item =>

        Form_Handler.Create_group_instances(
            opened_form_a       => opened_form,
            group               => update_add,
            number_of_instances => 1);

      when Inventory_Menus.update_change_item =>
        Form_Handler.Create_group_instances(
            opened_form_a       => opened_form,
            group               => update_change,
            number_of_instances => 1);

      when Inventory_Menus.update_delete_item =>
        Form_Handler.Create_group_instances(
            opened_form_a       => opened_form,
            group               => update_delete,
            number_of_instances => 1);

      when others =>
        null;

    end case;

    -- Open form's DDef for record access:
    --
    opened_record_form := Record_AM.Open_by_name(
        name         => update_form_pathname,
        input_output => Device_Defs.inout);

    loop
      -- Get a part ID:
      --
      form_status := Form_Handler.Get(
          opened_form_a   => opened_form,
          opened_window_a =>
              Inventory_Windows.main_window);

      if form_status /= Form_Defs.finished then

        -- Some kind of error processing
        null;

      else

        Form_Handler.Fetch_value(
            opened_form_a         => opened_form,
            element               => part_ID_field,
            subunit               => System_Defs.null_text,
              -- added subunit; value correct?
            value_buffer_VA       => part_ID'address,
            value_length          => part_ID'size/8,
            value_t               =>
                Data_Definition_Mgt.t_string,
```

```
    element_value_length => length,
    empty                => empty);

if empty then
  EXIT;         -- exit loop
else
  begin

    -- Get parts record, if possible:
    --
    new_part := false;

    Inventory_Files.Read_parts_record(
        part_ID      => part_ID,
        parts_record => parts_record);

    Record_AM.Ops.Update(
        opened_dev => opened_record_form,
        buffer_VA  => parts_record'address,
        length     => parts_record'size/8);

  exception
    when Inventory_Files.not_on_file =>
      new_part := true;

    when Inventory_Files.invalid_part_ID =>
      null;     -- "Invalid part ID
                -- entered" message has
                -- been displayed; go
                -- through loop again
  end;

  case selection is
    when Inventory_Menus.update_add_item =>
      if new_part then
        length := Record_AM.Ops.Read(
            opened_dev => opened_record_form,
            buffer_VA  => parts_record'address,
            length     => parts_record'size/8);

        Inventory_Files.Write_parts_record(
            parts_record => parts_record);

        -- Create and write log record:
        --
        Inventory_Files.Write_log_record(
          parts_record => parts_record,
          action       =>
              Inventory_Files.create);

      end if;

    when Inventory_Menus.update_change_item =>
      length := Record_AM.Ops.Read(
          opened_dev => opened_record_form,
          buffer_VA  =>
              new_parts_record'address,
          length     =>
              new_parts_record'size/8);

      Inventory_Files.Rewrite_parts_record(
          parts_record => parts_record);

      -- Create and write log record:
      --
      Inventory_Files.Write_log_record(
          parts_record => parts_record,
          action       =>
              Inventory_Files.update);

    when Inventory_Menus.update_delete_item =>

     Inventory_Files.Delete_parts_record(
        part_ID => part_ID);

      -- Create and write log record:
      --
```

```
            Inventory_Files.Write_log_record(
                parts_record => parts_record,
                action       =>
                    Inventory_Files.delete);

          when others =>
            null;

        end case;

      end if;    -- if not empty part ID

    end if;      -- if form finished

  end loop;

  Form_Handler.Clear(
      opened_form_a => opened_form);

  Form_Handler.Close_form(
      opened_form_a => opened_form);

  -- Close record access to form:
  --
  Record_AM.Ops.Close(
      opened_dev => opened_record_form);

end Process_update_form;


procedure Process_report_form(
    report_by_part:      boolean;
      -- True if by part, false if by location.
    report_out_dev:  out System_Defs.text)
      -- Returned output device's pathname,
      -- "System_Defs.null_text" if error.
  --
  -- Logic:
  --   1. Open report form
  --   2. Get report options and output device
  --   3. Attempt opening and closing report
  --      output device
  --   4. Clear and close form
  --   5. If any error occurred, return
  --      "report_out_dev" as "null_text"
is

  opened_form:  Form_Defs.opened_form_AD;
  form_status:  Form_Defs.status_t;

  length:       System.ordinal;
  empty:        boolean;

  report_options:  System.ordinal;
    -- Report options field value.

  valid:  boolean;
    -- Whether the report information is valid.

  test_out_dev:     System_Defs.text(Incident_Defs.txt_length);
    -- Entered report output device pathname to be
    -- checked.

  test_opened_dev:  Device_Defs.opened_device;
    -- Opened device returned from
    -- "Record_AM.Open" (test to see if
    -- entered device pathname is valid).

use Form_Defs;        -- import "/=" for type
                      -- "Form_Defs.status_t"

use System;      -- for "'size/8" arithmetic

begin

  opened_form := Get_form(report_form_pathname);

```

```
    form_status := Form_Handler.Get(
        opened_form_a   => opened_form,
        opened_window_a => Inventory_Windows.
                           main_window);

    if form_status /= Form_Defs.finished then

      -- some kind of error processing
      null;

    else

      Form_Handler.Fetch_value(
          opened_form_a        => opened_form,
          element              => rpt_type_field,
          subunit              => System_Defs.null_text,
            -- added subunit; value correct?
          value_buffer_VA      =>
              report_by_part'address,
          value_length         => report_by_part'size/8,
          value_t              =>
              Data_Definition_Mgt.t_boolean,
          element_value_length => length,
          empty                => empty);

      valid := not empty;


      Form_Handler.Fetch_value(
          opened_form_a        => opened_form,
          element              => rpt_opt_field,
          subunit              => System_Defs.null_text,
            -- added subunit; value correct?
          value_buffer_VA      =>
              report_options'address,
          value_length         => report_options'size/8,
          value_t              =>
              Data_Definition_Mgt.t_ord4,
          element_value_length => length,
          empty                => empty);


      valid := valid and (not empty);

      Form_Handler.Fetch_value(
          opened_form_a        => opened_form,
          element              => rpt_dev_field,
          subunit              => System_Defs.null_text,
            -- added subunit; value correct?
          value_buffer_VA      =>
              test_out_dev'address,
          value_length         => test_out_dev'size/8,
          value_t              =>
              Data_Definition_Mgt.t_string,
          element_value_length => length,
          empty                => empty);

      valid := valid and (not empty);


      -- Try to open device at the new pathname:
      --
      begin

        test_opened_dev := Record_AM.Open_by_name(
            name         => test_out_dev,
            input_output => Device_Defs.output);

        Record_AM.Ops.Close(
            opened_dev => test_opened_dev);

      exception

        when others =>
          valid := false;

          Message_Services.Write_msg(
```

```
                msg_id => invalid_output_device_code,
                param1 => Incident_Defs.message_parameter(
                    typ => Incident_Defs.txt,
                    len => test_out_dev.length)'(
                        typ     => Incident_Defs.txt,
                        len     => test_out_dev.length,
                        txt_val => test_out_dev));

        end; -- test open

      end if;  -- if form status = finished

      Form_Handler.Clear(
          opened_form_a => opened_form);

      Form_Handler.Close_form(
          opened_form_a => opened_form);

      if valid then
        report_out_dev := test_out_dev;
      else
        report_out_dev := System_Defs.null_text;
      end if;

    end Process_report_form;


  -- Form Processing Routine & Key Catchers:
  --

    procedure Validate_cost(
        old_parts_record:
            Inventory_Files.parts_record_type;
        qty_received:
            Inventory_Files.qty_type;
        total_cost:         in out
            Inventory_Files.cost_type;
        unit_cost:          in out
            Inventory_Files.cost_type;
        total:                     boolean := true;
        percentage_range:
            percentage_range_type := (5, 5);
        valid:                  out boolean)
    --
    -- Logic:
    --   Called from the Receipts form to validate unit
    --   cost and to calculate and return either total
    --   cost or unit cost.

    is

      max_cost, min_cost:  float;

    use System;   -- to import "*" and "/"

    begin

      -- Calculate total or unit cost:
      --
      if total then
        total_cost := float(unit_cost) *
                      float(qty_received);
      else
        unit_cost  := float(total_cost) /
                      float(qty_received);
      end if;

      -- Calculate minimum and maximum acceptable unit
      -- costs:
      --
      min_cost := float(old_parts_record.avg_unit_cost) *
          (1.0 - float(percentage_range.percent_less)
                 / 100.0);

      max_cost := float(old_parts_record.avg_unit_cost) *
          (1.0 + float(percentage_range.percent_less)
                 / 100.0);
```

```
    -- Check unit_cost against average cost:
    --
    valid := (unit_cost >= min_cost) and
             (unit_cost <= max_cost);

  end Validate_cost;



  procedure Go_to_inquiry
  --
  -- Logic:
  --   Called from the "Receipts" or "Change" form.
  --
  --   Calls "Process_inquiry_form".  Enclosing
  --   (calling) form is suspended during key-catcher
  --   call, resumed upon return from this procedure.
  is

  begin

    Process_inquiry_form;

  end Go_to_inquiry;


  procedure Add_supplier_ID(
      opened_form:  Form_Defs.opened_form_AD)
  --
  -- Logic:
  --   Called from the "Add" form.
  --
  --   Calls "Process_inquiry_form".  Enclosing
  --   (calling) form is suspended during key-catcher
  --   call, resumed upon return from this procedure.
  is

  begin

    begin
      -- Add another instance of the supplier ID group.
      Form_Handler.Create_group_instances(
          opened_form_a       => opened_form,
          group               => suppliers_field,
          number_of_instances => 1);

    exception
      when Form_Handler.maximum_number_reached => null;

    end;

  end Add_supplier_ID;

end Inventory_Forms;
```

X-A.5.6 Inventory_Menus Package Specification

```
with Device_Defs,
     Incident_Defs,
     Inventory_Messages,
     System,
     System_Defs,
     Terminal_Defs,
     Window_Services;

package Inventory_Menus is
  --
  -- Function:
  --   Contains subprograms to install and process
  --   Inventory Example Program menus.
  --
  --   This package contains the routines which
  --   perform each menu's selection actions.  Some of
  --   the menu selections require calls to the
  --   "Inventory_Forms" and "Inventory_Reports"
  --   packages.
  --
  -- History:
  --   05-18-87, William A. Rohm:  Written.
  --   10-27-87, WAR:              Revised.
  --
  -- End of Header


  -- Incident codes for messages:
  --
  module:  constant := 4;
    -- Message module index.

  -- *M*   set.language  :language = English
  -- *M*   create.variable  module  :value = 4

  unable_to_install_code:  constant
      Incident_Defs.incident_code := (
           message_object =>
               Inventory_Messages.message_object,
           module              => module,
           number              => 0,
           severity            =>
               Incident_Defs.error);

  -- *M*   store  :module = $module  :number = 0\
  -- *M*          :msg_name = unable_install \
  -- *M*          :short = "Unable to install menus."


  no_selection_code:  constant
      Incident_Defs.incident_code := (
           message_object =>
               Inventory_Messages.message_object,
           module              => module,
           number              => 1,
           severity            =>
               Incident_Defs.warning);

  -- *M*   store  :module = $module  :number = 1\
  -- *M*          :msg_name =  no_selection\
  -- *M*          :short = "Selection $p1<selection
  -- *M*                    number> is not implemented."


  menu_group_DDef_path:
      System_Defs.text(34) := (34,34,
        "/examples/inventory/DDef/menu_DDef");
    -- Pathname of stored menu group DDef.

  menu_group_DDef_root_name:
      System_Defs.text(4) := (4,4,"root");
    -- Pathname of menu group DDef's root node.

  inv_menu_group_ID:  constant
```

```
        Terminal_Defs.menu_group_ID := 1;
      -- Inventory menu group's ID.


-- Menu IDs
   inquiry_menu_ID:        constant
       Terminal_Defs.menu_ID := 1;

   posting_menu_ID:        constant
       Terminal_Defs.menu_ID := 2;

   update_menu_ID:         constant
       Terminal_Defs.menu_ID := 3;

   report_menu_ID:         constant
       Terminal_Defs.menu_ID := 4;

   housekeeping_menu_ID:   constant
       Terminal_Defs.menu_ID := 5;

   exit_menu_ID:           constant
       Terminal_Defs.menu_ID := 6;

-- Inquiry menu items
   inq_by_part_item:  constant
       Terminal_Defs.menu_item_ID := 1;
   inq_by_desc_item:  constant
       Terminal_Defs.menu_item_ID := 2;
   inq_exit_item:     constant
       Terminal_Defs.menu_item_ID := 3;


-- Posting menu items
   post_receipt_item:    constant
       Terminal_Defs.menu_item_ID := 1;
   post_issue_item:      constant
       Terminal_Defs.menu_item_ID := 2;
   post_return_item:     constant
       Terminal_Defs.menu_item_ID := 3;
   post_spoilage_item:   constant
       Terminal_Defs.menu_item_ID := 4;
   post_journal_item:    constant
       Terminal_Defs.menu_item_ID := 5;
   post_exit_item:       constant
       Terminal_Defs.menu_item_ID := 6;


-- Update menu items
   update_add_item:      constant
       Terminal_Defs.menu_item_ID := 1;
   update_change_item:   constant
       Terminal_Defs.menu_item_ID := 2;
   update_delete_item:   constant
       Terminal_Defs.menu_item_ID := 3;
   update_exit_item:     constant
       Terminal_Defs.menu_item_ID := 4;


-- Report menu items
   report_by_part_item:       constant
       Terminal_Defs.menu_item_ID := 1;
   report_by_location_item:   constant
       Terminal_Defs.menu_item_ID := 2;
   report_exit_item:          constant
       Terminal_Defs.menu_item_ID := 3;


-- Housekeeping menu items
   hskpg_index_item:  constant
       Terminal_Defs.menu_item_ID := 1;
   hskpg_exit_item:   constant
       Terminal_Defs.menu_item_ID := 2;


   procedure Set_up_menu_group;
     --
     -- Function:
```

```
    --   Retrieve Inventory Example Program's menu
    --   group description (*a menu DDef*), then
    --   install and enable the menu group in the main
    --   window.


-- Menu selection processing procedures:
--   Inquiry / Posting / Update / Report / Housekeeping
--

  procedure Process_inquiry_menu(
      selection:  Terminal_Defs.menu_item_ID);
        -- Selection made in this menu.
    --
    -- Function:
    --   Processes selections from the Inquiry menu.



  procedure Process_posting_menu(
      selection:  Terminal_Defs.menu_item_ID);
        -- Selection made in this menu.
    --
    -- Function:
    --   Processes selections from the Posting menu.



  procedure Process_update_menu(
      selection:  Terminal_Defs.menu_item_ID);
        -- Selection made in this menu.
    --
    -- Function:
    --   Processes selections from the Update menu.



  procedure Process_report_menu(
      selection:  Terminal_Defs.menu_item_ID);
        -- Selection made in this menu.
    --
    -- Function:
    --   Processes selections from the Report menu.


  procedure Process_housekeeping_menu(
      selection:  Terminal_Defs.menu_item_ID);
        -- Selection made in this menu.
    --
    -- Function:
    --   Processes selections from the Housekeeping
    --   menu.

end Inventory_Menus;
```

X-A.5.7 Inventory_Menus Package Body

```
with Data_Definition_Mgt,
     Device_Defs,
     Directory_Mgt,
     File_Admin,
     File_Defs,
     Incident_Defs,
     Inventory_Files,
     Inventory_Forms,
     Inventory_Messages,
     Inventory_Reports,
     Inventory_Windows,
     Message_Services,
     Record_AM,
     System_Defs,
     Terminal_Defs,
     Unchecked_Conversion,
     Window_Services;

package body Inventory_Menus is

  -- Generic function:
  --
    function DDef_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Data_Definition_Mgt.DDef_AD);

  -- Variables:
  --
  menu_group_DDef_AD:  Data_Definition_Mgt.DDef_AD;
    -- AD to stored menu group DDef.

  menu_group_node:
      Data_Definition_Mgt.node_reference;
    -- Node reference to stored menu group DDef.


  procedure Set_up_menu_group

  is

  begin

    -- Retrieve menu group's DDef:
    --
    menu_group_DDef_AD := DDef_from_untyped(
        Directory_Mgt.Retrieve(
            name => menu_group_DDef_path));


    -- Retrieve menu group's root node:
    --
    menu_group_node := Data_Definition_Mgt.
        Retrieve_DDef(
            DDef => menu_group_DDef_AD,
            name => menu_group_DDef_root_name);


        -- Install menu group:
        --
        Window_Services.Ops.Install_menu_group(
          window     => Inventory_Windows.
                        main_window,
          menu_group => menu_group_node,
          ID         => inv_menu_group_ID);

        -- Enable menu group:
        --
        Window_Services.Ops.Menu_group_enable(
          window     => Inventory_Windows.
                        main_window,
          menu_group => inv_menu_group_ID,
          enable     => true);

```

```
end Set_up_menu_group;



procedure Process_inquiry_menu(
    selection:  Terminal_Defs.menu_item_ID)
      -- Selection made in this menu.
is
-- Logic:
--   Determine item selection, perform actions.

begin

  case selection is

    when inq_by_part_item => Inventory_Forms.
        Process_inquiry_form;

    when inq_by_desc_item =>

      Message_Services.Write_msg(
          msg_id => no_selection_code,
          param1 =>
              Incident_Defs.message_parameter(
              typ => Incident_Defs.ord,
              len => 0)'(
                  typ   => Incident_Defs.ord,
                  len   => 0,
                  o_val => selection));

    when inq_exit_item =>
      return;

    when others => null;

  end case;

end Process_inquiry_menu;



procedure Process_posting_menu(
    selection:  Terminal_Defs.menu_item_ID)
      -- Selection made in this menu.
is
-- Logic:
--   Determine item selection, perform actions.

begin
  case selection is
    when post_receipt_item => Inventory_Forms.
        Process_receipts_form;

    when post_issue_item    |
         post_return_item   |
         post_spoilage_item |
         post_journal_item =>

      Message_Services.Write_msg(
          msg_id => no_selection_code,
          param1 => Incident_Defs.message_parameter(
              typ => Incident_Defs.ord,
              len => 0)'(
                  typ   => Incident_Defs.ord,
                  len   => 0,
                  o_val => selection));

     when post_exit_item =>
       return;

     when others => null;

   end case;
 end Process_posting_menu;
```

```
procedure Process_update_menu(
    selection:  Terminal_Defs.menu_item_ID)
                  -- Selection made in this menu.
is
-- Logic:
--   Determine item selection, perform actions.

begin

  case selection is

    when update_add_item    |
         update_change_item |
         update_delete_item =>

      Inventory_Forms.Process_update_form(
          selection => selection);

     when update_exit_item =>
       return;

     when others => null;

   end case;

 end Process_update_menu;



procedure Process_report_menu(
    selection:  Terminal_Defs.menu_item_ID)
      -- Selection made in this menu.
is

  report_out_dev:  System_Defs.text(256);

begin

  case selection is

    when report_by_part_item =>

        Inventory_Forms.Process_report_form(
            report_by_part => true,
            report_out_dev => report_out_dev);

        Inventory_Reports.Print_report_by_part(
            output_dev_pathname => report_out_dev);


    when report_by_location_item =>

        Inventory_Forms.Process_report_form(
            report_by_part => false,
            report_out_dev => report_out_dev);

        Inventory_Reports.Print_report_by_location(
            output_dev_pathname => report_out_dev);

    when report_exit_item =>
      return;

    when others => null;

  end case;

end Process_report_menu;



procedure Process_housekeeping_menu(
    selection:  Terminal_Defs.menu_item_ID)
      -- Selection made in this menu.
is

begin

```

```
    case selection is

      when hskpg_index_item =>

        File_Admin.Reorganize_index(
            file  => File_Defs.Convert_device_to_file(
                s => Record_AM.Ops.Get_device_object(
                    opened_dev =>
                        Inventory_Files.parts_file)),
            index =>
                Inventory_Files.part_ID_index_name);

      when hskpg_exit_item  =>
        return;

      when others => null;

    end case;

  end Process_housekeeping_menu;


end Inventory_Menus;
```

X-A.5.8 Inventory_Reports Package Specification

```
with Device_Defs,
     Incident_Defs,
     Inventory_Messages,
     System,
     System_Defs,
     Terminal_Defs,
     Window_Services;

package Inventory_Reports is
  --
  -- Function:
  --   Contains two procedures to process and
  --   print either of the Inventory Program
  --   reports (by part ID solely, or by part
  --   location and then part ID) from the
  --   Inventory Parts file.
  --
  --   One or the other of these procedures is
  --   called from the Report Menu by the
  --   apppropriate menu selection: "Print
  --   "Report by Part", or "Print Report by"
  --   "Location".
  --
  -- History:
  --   05-21-87,  William A. Rohm:  Written.
  --   10-27-87,  WAR:              Revised.
  --
  -- End of Header

  -- Incident codes for messages:
  --
  module:  constant := 6;
    -- Message module index.

  --*M*   set.language  :language = English
  --*M*   create.variable  module  :value = 6

  report_printing_code:  constant
      Incident_Defs.incident_code := (
          message_object =>
              Inventory_Messages.message_object,
          module         => module,
          number         => 0,
          severity       =>
              Incident_Defs.information);

  --*M*   store  :module = $module  :number = 0\
  --*M*          :msg_name = report_printing \
  --*M*          :short = "Inventory parts file
  --*M*             report by $p1<part/location>
  --*M*             is now printing on device
  --*M*             $p2<output device name>."


  report_by_part_DDef_str:  constant string :=
      "/example/inventory/DDefs/report_by_part";
    -- String constant for "report by part"
    -- report DDef's pathname.

  report_by_part_DDef_pathname:
      System_Defs.text(
          report_by_part_DDef_str'length) := (
              report_by_part_DDef_str'length,
              report_by_part_DDef_str'length,
              report_by_part_DDef_str);
    -- Text constant from "report by part"
    -- DDef's pathname string.


  report_by_loc_DDef_str:  constant string :=
  "/example/inventory/DDefs/report_by_location";
    -- String constant for "report by location"
    -- report DDef's pathname.

```

```
   report_by_loc_DDef_pathname:
       System_Defs.text(
           report_by_loc_DDef_str'length) := (
               report_by_loc_DDef_str'length,
               report_by_loc_DDef_str'length,
               report_by_loc_DDef_str);
     -- Text constant from "report by location"
     -- DDef's pathname string.


   sort_by_loc_DDef_str:  constant string :=
   "/example/inventory/DDefs/sort_by_location";
     -- String constant for "sort by location"
     -- "(then by part ID)" sort DDef's pathname.

   sort_by_loc_DDef_pathname:
       System_Defs.text(
           sort_by_loc_DDef_str'length) := (
               sort_by_loc_DDef_str'length,
               sort_by_loc_DDef_str'length,
               sort_by_loc_DDef_str);
     -- Text constant from "sort by location"
     -- DDef's pathname string.


   procedure Print_report_by_part(
       output_dev_pathname:  System_Defs.text);
          -- Pathname of output device for
          -- printing report.  Can be any device
          -- supporting the byte stream access
          -- method.
     --
     -- Function:
     --   Prepares report *by part ID* from parts
     --   file, then prints report to given
     --   output device.


   procedure Print_report_by_location(
       output_dev_pathname:  System_Defs.text);
          -- Pathname of output device for
          -- printing report.  Can be any device
          -- supporting the byte stream access
          -- method.
     --
     -- Function:
     --   Sorts parts file by location (and then
     --   by part ID) into temporary file, then
     --   prints report to given output device.

end Inventory_Reports;
```

X-A.5.9 `Inventory_Reports` Package Body

Note: This example could not be compiled successfully due to the absence of the the `Report_Handler` package at the time of this printing.

```
with Byte_Stream_AM,
     Data_Definition_Mgt,
     Device_Defs,
     Directory_Mgt,
     Event_Mgt,
     File_Admin,
     File_Defs,
     Incident_Defs,
     Inventory_Files,
     Inventory_Windows,
     Message_Services,
     Passive_Store_Mgt,
     Pipe_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     Record_AM,
     Report_Handler,
     Sort_Merge_Interface,
     System,
     System_Defs,
     Terminal_Defs,
     Unchecked_conversion,
     Volume_Set_Defs;

package body Inventory_Reports is
  --
  -- History:
  --   05-21-87,  William A. Rohm:  Written.
  --   10-27-87,  WAR:              Revised.
  --
  -- End of Header

  -- Generic function:
  --
  function DDef_from_untyped is new
      Unchecked_conversion(
          source => System.untyped_word,
          target => Data_Definition_Mgt.DDef_AD);

  -- Type:
  --
  type connection_record is
    -- Defines sort pipe's input and output, for
    -- "Sort" and "Print" processes (called by
    -- "Print_report_by_location").
    record
      sort_out:     Device_Defs.opened_device;
        -- Output from "Sort" to pipe.
      report_in:   Device_Defs.opened_device;
        -- Input from pipe to "Print".
      report_out:   Device_Defs.opened_device;
        -- Output device for "Print".
    end record;


  procedure Print_report_by_part(
      output_dev_pathname:  System_Defs.text)
    --
    -- Logic:
    --   1. Open parts file for reading
    --   2. Open report output device
    --   3. Get report DDef and initialize report
    --   4. Print report and display message

  is

  opened_output:  Device_Defs.opened_device;
    -- Opened output device for printing report.

  report_DDef:  Data_Definition_Mgt.DDef_AD;
    -- AD to a report data definition.
```

```
initialized_report:  Device_Defs.opened_device;
  -- Initialized (opened) report object itself.

  local_parts_file:  Device_Defs.device :=
      Record_AM.Ops.Get_device_object(
          Inventory_Files.parts_file);
   -- AD to parts file.

  opened_local_parts_file:
      Device_Defs.opened_device;
    -- AD to locally opened parts file.

  part:  System_Defs.text(4) := (4,4,"part");
    -- Parameter to "report_printing" message,
    -- since this report is by "part".

begin

  -- Open parts file for reading, so no
  -- concurrent updates will interfere:
  --
  opened_local_parts_file := Record_AM.Ops.Open(
      dev          => local_parts_file,
      input_output => Device_Defs.input,
      allow        => Device_Defs.readers);


  -- Open output device:
  --
  opened_output := Byte_Stream_AM.Open_by_name(
      name         =>
          output_dev_pathname,
      input_output =>
          Device_Defs.output);


  -- Get report definition (DDef):
  --
  report_DDef := DDef_from_untyped(
     Directory_Mgt.Retrieve(
         name => report_by_part_DDef_pathname));
  -- Assume "Report_Handler.Is_report".


  -- Initialize report:
  --
  initialized_report := Report_Handler.Initialize(
      description => report_DDef,
      input       => opened_local_parts_file,
      output      => opened_output);


  -- Print report:
  --
  Report_Handler.Print(
      report => initialized_report);


  -- Display "report_printing" message:
  --
  Message_Services.Write_msg(
      msg_id => report_printing_code,
      param1 => Incident_Defs.message_parameter(
          typ => Incident_Defs.txt,
          len => part.length)'(
              typ     => Incident_Defs.txt,
              len     => part.length,
              txt_val => part),
      param2 => Incident_Defs.message_parameter(
          typ => Incident_Defs.txt,
          len => output_dev_pathname.length)'(
              typ   => Incident_Defs.txt,
              len   => output_dev_pathname.length,
              txt_val => output_dev_pathname),
      device => Inventory_Windows.message_window);

```

```

  -- Close locally opened parts file:
  --
  Record_AM.Ops.Close(
      opened_dev => opened_local_parts_file);

end Print_report_by_part;


procedure Sort(
  param_buffer:  System.address;
    -- Address of connection record.
  param_length:  System.ordinal)
    -- Not used in this procedure, but required for
    -- process's initial procedure.
--
-- Logic:
--   1. Open local copy of parts file (sort input)
--   2. Get sort DDef and perform sort
is

  conn_rec:  connection_record;
    -- Record containing pipe input/output devices.
    FOR conn_rec USE AT param_buffer;

  local_parts_file:  Device_Defs.device :=
      Record_AM.Ops.Get_device_object(
          Inventory_Files.parts_file);
   -- AD to parts file.

  opened_local_parts_file:  Device_Defs.opened_device;
    -- AD to locally opened parts file.

  opened_sort_DDef:
      Device_Defs.opened_device;
  sort_DDef_reference:
      Data_Definition_Mgt.node_reference;

begin

  -- Open parts file for reading, so no
  -- concurrent updates will interfere:
  --
  opened_local_parts_file := Record_AM.Ops.Open(
      dev          => local_parts_file,
      input_output => Device_Defs.input,
      allow        => Device_Defs.readers);


  -- Open sort definition (DDef):
  --
  opened_sort_DDef := Record_AM.Open_by_name(
      name         =>
          sort_by_loc_DDef_pathname,
      input_output => Device_Defs.input,
      allow        => Device_Defs.readers,
      block        => true);

  -- Get sort DDef's node reference:
  --
  sort_DDef_reference :=
      Record_AM.Ops.Get_DDef(
          opened_dev => opened_sort_DDef);


  -- Perform sort, using sort DDef, from parts
  -- file to pipe:
  --
  Sort_Merge_Interface.Sort(
      input_device  =>
          opened_local_parts_file,
      DDef          => sort_DDef_reference,
      output_device => conn_rec.sort_out,
      stable_sort   => true,
      tuning_opts   =>
          Sort_Merge_Interface.no_tuning);

```

```
    -- Close locally opened parts file:
    --
    Record_AM.Ops.Close(
        opened_dev => opened_local_parts_file);

  end Sort;
    pragma subprogram_value(
        Process_Mgt.Initial_proc,
        Sort);


  procedure Print(
    param_buffer:  System.address;
      -- Address of connection record.
    param_length:  System.ordinal)
      -- Not used in this procedure, but required for
      -- process's initial procedure.
    --
    -- Logic:
    --   1. Get report DDef
    --   2. Open report output
    --   3. Get report DDef and initialize report
    --   4. Print report from pipe output.

  is

    report_DDef:  Data_Definition_Mgt.DDef_AD;
      -- AD to a report data definition.

    initialized_report:  Device_Defs.opened_device;
      -- Initialized (opened) report object itself.

    conn_rec:  connection_record;
      -- Record containing pipe input/output devices.
      FOR conn_rec USE AT param_buffer;

  begin

    -- Get report definition (DDef):
    --
    report_DDef := DDef_from_untyped(
        Directory_Mgt.Retrieve(
            report_by_loc_DDef_pathname));

    -- Initialize report:
    --
    initialized_report := Report_Handler.Initialize(
        description => report_DDef,
        input       => conn_rec.report_in,
        output      => conn_rec.report_out);

    -- Print report:
    --
    Report_Handler.Print(
        report => initialized_report);

    -- Close report output device:
    --
    Record_AM.Ops.Close(
        opened_dev => conn_rec.report_out);

  end Print;
    pragma subprogram_value(Process_Mgt.Initial_proc,
                            Print);


  procedure Print_report_by_location(
      output_dev_pathname:  System_Defs.text)
    --
    -- Logic:
    --   1. Open pipe input (sort output) and
    --      output (report input)
    --   2. Spawn "Sort" and "Print" processes
    --   3. Wait for termination of processes
    --   4. Deallocate processes
    --   5. Display "report printing" message
  is
```

```

    conn_rec:  connection_record;
      -- Record referencing all I/O connections used by
      -- the child processes.

    sort_pipe:  Pipe_Mgt.pipe_AD;
      -- Pipe from sort output to report input.

    this_process_untyped:  System.untyped_word;
      -- Process executing call to
      -- "Print_report_by_location", as an
      -- untyped word.

    sort_process:  Process_Mgt_Types.process_AD;
      -- Process executing "Sort".

    print_process:  Process_Mgt_Types.process_AD;
      -- Process executing "Print".

    term_events:  Event_Mgt.action_record_list(2);
      -- Array that receives termination events of the
      -- two child processes.

    location:  System_Defs.text(8) := (8,8,"location");
      -- Parameter to "report_printing" message, since
      -- this report is by "location".

  begin

    -- Create pipe:
    --
    sort_pipe := Pipe_Mgt.Create_pipe;

    -- Open sort output, report input, and report
    -- output devices:
    --
    conn_rec := (
        sort_out    => Record_AM.Ops.Open(
            Pipe_Mgt.Convert_pipe_to_device(
                sort_pipe),
            Device_Defs.output),
        report_in   => Record_AM.Ops.Open(
            Pipe_Mgt.Convert_pipe_to_device(
                sort_pipe),
            Device_Defs.input),
        report_out  => Record_AM.Open_by_name(
            output_dev_pathname,
            Device_Defs.output));

    -- Get this process's AD:
    --
    this_process_untyped :=
        Process_Mgt.Get_process_globals_entry(
            Process_Mgt_Types.process);

    -- Spawn "Sort" process:
    --
    sort_process := Process_Mgt.Spawn_process(
        init_proc    => Sort'subprogram_value,
        param_buffer => conn_rec'address,
        term_action  => (
            event       => Event_Mgt.user_1,
            message     => System.null_address,
            destination => this_process_untyped));

    -- Spawn "Print" process:
    --
    print_process := Process_Mgt.Spawn_process(
        init_proc    => Print'subprogram_value,
        param_buffer => conn_rec'address,
        term_action  => (
            event       => Event_Mgt.user_2,
            message     => System.null_address,
            destination => this_process_untyped));

    -- Wait for both processes to finish:
    --
```

```
      Event_Mgt.Wait_for_all(
          events =>
              (Event_Mgt.user_1 .. Event_Mgt.user_2 =>
                   true,
               others => false),
          action_list => term_events);

      -- The two processes must have terminated, so they
      -- can be deallocated:
      --
      Process_Mgt.Deallocate(sort_process);
      Process_Mgt.Deallocate(print_process);

      -- Display "report printing" message:
      --
      Message_Services.Write_msg(
        msg_id => report_printing_code,
        param1 => Incident_Defs.message_parameter(
          -- "location"
            typ => Incident_Defs.txt,
            len => location.length)'(
                typ     => Incident_Defs.txt,
                len     => location.length,
                txt_val => location),
        param2 => Incident_Defs.message_parameter(
          -- "output device pathname"
            typ => Incident_Defs.txt,
            len => output_dev_pathname.length)'(
                typ   => Incident_Defs.txt,
                len   => output_dev_pathname.length,
                txt_val => output_dev_pathname));

    end Print_report_by_location;

end Inventory_Reports;
```

X-A.5.10 `Inventory_Windows` Package Specification

```
with Device_Defs,
     Terminal_Defs;

package Inventory_Windows is
  --
  -- Function:
  --   Contains procedures to open and close the two
  --   Inventory Program windows:  the main window and
  --   the message window.
  --
  --   The main window is used for menu and form
  --   display and for user data entry.  The message
  --   window is only used to display status and error
  --   messages to the user.
  --
  -- History:
  --   06-04-87, William A. Rohm:  Written.
  --
  -- End of Header

  -- Constants:
  --
    module:  constant := 2;
      -- Message module index value, for this
      -- package's messages.  Not currently used.

    main_window_size:  Terminal_Defs.point_info := (
        80,20);
      -- Size of main window, in columns and rows.

    main_buffer_size:  Terminal_Defs.point_info := (
        80,20);
      -- Size of main window's buffer.

    main_window_pos:  Terminal_Defs.point_info := (
        1,1);
      -- Position of main window (upper left corner).

    message_window_size:  Terminal_Defs.point_info := (
        80,3);
      -- Size of message window, in columns and rows.

    message_buffer_size:  Terminal_Defs.point_info := (
        80,3);
      -- Size of message window's buffer.

    message_window_pos:  Terminal_Defs.point_info := (
        1, 1 + main_window_pos.vert);
      -- Position of message window (just below main
      -- window).

  -- Variables:
  --
    main_window:  Device_Defs.opened_device;
      -- Main window, for displaying menus and forms
      -- and getting user input.  Usable by other
      -- modules after "Open_program_windows" has been
      -- called.

    message_window:  Device_Defs.opened_device;
      -- Message window, for status and error
      -- messages.  Usable by other modules after
      -- "Open_program_windows" has been called.



  procedure Open_program_windows;
    --
    -- Function:
    --   Open both program windows (main and message)
    --   on the current terminal.
    --
    --   The main window is for the Inventory
    --   Program's menus and forms.  The message
```

```
      --   window is opened, for message display.
      --
      --   The main window is opened at the top of the
      --   screen.  The message window is opened below
      --   the main window.



   procedure Close_program_windows;
      --
      -- Function:
      --   Closes both Inventory Program windows:  main
      --   window and message window.


end Inventory_Windows;

```

X-A.5.11 Inventory_Windows Package Body

```
with Byte_Stream_AM,
     Device_Defs,
     Process_Mgt,
     Process_Mgt_Types,
     System,
     Terminal_Defs,
     Window_Services;

package body Inventory_Windows
  is

  procedure Open_program_windows
    --
    -- Logic:
    --  1.  Gets device AD to underlying terminal.
    --  2.  Opens main window, assigning
    --          "inventory_main".
    --  3.  Opens message window, assigning
    --          "inventory_message".

  is
        old_opened_window:    Device_Defs.opened_device;
        old_window:           Device_Defs.device;
        underlying_terminal:  Device_Defs.device;

  begin

    -- Assume standard input, on entry, is from an
    -- opened window:
    --
    old_opened_window :=
        Process_Mgt.Get_process_globals_entry(
            Process_Mgt_Types.standard_input);

    -- Get device object of standard input window:
    --
    old_window := Byte_Stream_AM.Ops.Get_device_object(
        old_opened_window);

    -- Get device AD of standard input window's
    -- terminal:
    --
    underlying_terminal :=
        Window_Services.Ops.Get_terminal(
            old_window);

    -- Create new main window:
    --
    main_window := Window_Services.Ops.Create_window(
        terminal            => underlying_terminal,
        pixel_units         => false,
          -- characters, not pixels
        fb_size             => main_buffer_size,
        desired_window_size => main_window_size,
        window_pos          => main_window_pos,
        view_pos            =>
            Terminal_Defs.point_info'(1,1));


    -- Create new message window:
    --
    message_window := Window_Services.Ops.Create_window(
        terminal            => underlying_terminal,
        pixel_units         => false,
        fb_size             => message_buffer_size,
        desired_window_size => message_window_size,
        window_pos          => message_window_pos,
        view_pos            =>
            Terminal_Defs.point_info'(1,1));

  end Open_program_windows;
```

```
  procedure Close_program_windows
    --
    -- Logic:
    --   1. Closes main window.
    --   2. Closes message window.

  is

  begin

    Window_Services.Ops.Destroy_window(main_window);

    Window_Services.Ops.Destroy_window(message_window);

  end Close_program_windows;

end Inventory_Windows;
```

X-A.5.12 `Inventory_Messages` Package Specification

```
with Incident_Defs,
     System,
     System_Defs;

package Inventory_Messages is
  --
  -- Function:
  --   Defines Inventory Example Program's message
  --   object, used for all incident code declarations
  --   in the program.
  --
  --   Each package defines its own messages (using
  --   tagged message definitions) with its unique
  --   module number.
  --
  -- History:
  --   07-27-87, William A. Rohm:  Written.
  --   10-27-87, WAR:              Revised.
  --
  -- End of Header

  -- Constants:
  --
  message_file:  constant System_Defs.text_AD :=
      new System_Defs.text'(
          31,31,"/example/inventory/message_file");
    -- AD to message file text name.
    --
    -- *This will go away when "pragma bind" changes.*


  message_object:  constant System.untyped_word :=
      System.null_word;

    pragma bind (message_object,
                "inventory_messages.message_file");
   -- Message object for Inventory Program incident
   -- codes.  Bound to "message_file" constant by
   -- pragma "bind".
   --
   -- *When the resident compiler/linker is in place,*
   -- *this pragma will become:*
   -- |  pragma bind(message_object,
   -- |               "/example/inventory/message_file");

end Inventory_Messages;
```

X-A.6 Program Services

X-A.6.1 `At_cmd_ex` Procedure

```
with At_Support_Ex,
     Command_Handler,
     Device_Defs,
     Long_Integer_Defs,
     Message_Services,
     System_Defs,
     Timed_Requests_Mgt;

procedure At_cmd_ex
  --
  -- Function:
  --   This procedure will run a command at a specified time.
  --   It sets defaults for unspecified parameters and
  --   parses mandatory and specified time parameters
  --   and calls subprogram that will initial a new session
  --   and job to run the command.  The prompt will
  --   return after the new job is started.  The until
  --   and count arguments are only effective if period is
  --   set
  --
  -- History:
  --   04-05-88, Ed Sassone, creation date
  --   05-20-88, Ed Sassone, working version
  --
  -- End of Header
  --
  --
  -- Command Definition:
  -- at_cmd_ex     :time=<extended_string_list(1..25(1..11))>
  --               :command=<extended_string(1..80)>
  --               [:period=<extended_string_list(0..25(0..11))>:=("()")]
  --               [:until=<extended_string_list(0..25(0..11))>:=("()")]
  --               [:count=<integer(1..1_000)>:=1_000]
  --
  --
  --*D*  manage.commands
  --*D*    create.invocation_command
  --*D*
  --*D*      define.argument time \
  --*D*        :type = string_list
  --*D*        set.maximum_length 25 11
  --*D*        set.mandatory
  --*D*      end
  --*D*
  --*D*      define.argument command \
  --*D*        :type = string
  --*D*        set.maximum_length 80
  --*D*        set.mandatory
  --*D*      end
  --*D*
  --*D*      define.argument period \
  --*D*        :type = string_list
  --*D*        set.maximum_length 25 11
  --*D*        allow.null_values :list :element
  --*D*        set.value_default "()"
  --*D*      end
  --*D*
  --*D*      define.argument until \
  --*D*        :type = string_list
  --*D*        set.maximum_length 25 11
  --*D*        allow.null_values :list :element
  --*D*        set.value_default "()"
  --*D*      end
  --*D*
  --*D*      define.argument count \
  --*D*        :type=integer
  --*D*        set.value_default 1000       -- function ($$upper) NYI
  --*D*        set.bounds 1..1000           -- open bounds NYI
  --*D*      end
  --*D*
  --*D*    end           -- create.invocation_command

is
```

```

   use Long_Integer_Defs;       -- for time comparison


   odo:        Device_Defs.opened_device;

   -- parameters
   time:        System_Defs.string_list(25) :=
       (25, 0, 0,  (others => ' '));

   command:    System_Defs.text(80) :=
       (80, 0, (others => ' '));

   period:     System_Defs.string_list(25) :=
       (25, 0, 0, (others => ' '));

   until:      System_Defs.string_list(25) :=
       (25, 0, 0, (others => ' '));

   count:      integer;


   start_at:   System_Defs.system_time_units :=
       System_Defs.null_time;
     -- stu equivalent of time

   next_at:    System_Defs.system_time_units :=
       System_Defs.null_time;
     -- stu equivalent of period

   until_at:   System_Defs.system_time_units :=
       Long_Integer_Defs.max_int;
     -- stu equivalent of until

begin

  odo :=  Command_Handler.
      Open_invocation_command_processing;

  Command_Handler.Get_string_list(
      cmd_odo    => odo,
      arg_number => 1,
      arg_value  => time);

  Command_Handler.Get_string(
      cmd_odo    => odo,
      arg_number => 2,
      arg_value  => command);

  Command_Handler.Get_string_list(
      cmd_odo    => odo,
      arg_number => 3,
      arg_value  => period);

  Command_Handler.Get_string_list(
      cmd_odo    => odo,
      arg_number => 4,
      arg_value  => until);

  count :=
      Command_Handler.Get_integer(
          cmd_odo    => odo,
          arg_number => 5);

  Command_Handler.Close(odo);

    -- parse timing arguments

    start_at :=
        At_Support_Ex.Parse_time(
            time      => time,
            from_when => Timed_Requests_Mgt.system_epoch);

    if period.length > 4 then
      -- keep defaults if nothing assigned
      next_at :=
          At_Support_Ex.Parse_time(
```

```
              time      => period,
              from_when => Timed_Requests_Mgt.now);
    else
      count := 1;  -- if no period do command only once
    end if;

    if until.length > 4 then
      -- keep defaults if nothing assigned
      until_at :=
          At_Support_Ex.Parse_time(
              time      => until,
              from_when => Timed_Requests_Mgt.system_epoch);
    end if;

   if start_at < Timed_Requests_Mgt.get_time then

     Message_Services.Write_msg(
         msg_id => At_Support_Ex.prior_time_warning_code);
   end if;

   -- creates new session and job so prompt will return
   At_Support_Ex.Create_waiting_process(
      invocation_record => At_Support_Ex.program_record'(
          command    => command,
          stu_start  => start_at,
          stu_period => next_at,
          stu_until  => until_at,
          count      => count));

end At_cmd_ex;
```

X-A.6.2 At_Support_Ex Package Specification

```
with Incident_Defs,
     Process_Mgt,
     Timed_Requests_Mgt,
     System,
     System_Defs;

package At_Support_Ex is
  --
  -- Function:
  --  Provides support for At_cmd_ex.  Parses time
  --  arguments and invokes the given command either
  --  once at the specified time or from the given time
  --  multiple times based on a specified period until
  --  a given count or time limit, whichever is first.
  --
  -- History:
  --  04-05-88, Ed Sassone, creation date
  --  05-20-88, Ed Sassone, working version
  --
  -- Exception Codes:
  msg_obj: constant System.untyped_word :=
           System.null_word;    -- use oeo

  time_format_error_code: constant Incident_Defs.
      incident_code := (
          module         => 0,
          number         => 1,
          severity       => Incident_Defs.error,
          message_object => msg_obj);
  day_format_error_code: constant Incident_Defs.
      incident_code := (
          module         => 0,
          number         => 2,
          severity       => Incident_Defs.error,
          message_object => msg_obj);

  prior_time_warning_code: constant Incident_Defs.
      incident_code := (
          module         => 0,
          number         => 3,
          severity       => Incident_Defs.warning,
          message_object => msg_obj);

  --
  -- Exceptions:
  --
  --*D* manage.messages
  --
  time_format_error: exception;
  -- Occurs when the time was not input in a proper
  -- format
  --*D*   store 0 1 time_format_error \
  --*D*   :short = "$p1 is an improper time specification
  --*D*The correct format is hh[:mm[:ss[.dd]]]"
  day_format_error: exception;
  -- Occurs when the day was not input in a proper
  -- format
  --*D*   store 0 2 day_format_error \
  --*D*   :short = "$p1 is an improper time specification
  --*D*The correct format is  [MM/]DD[/YYYY]]"


  -- Warning message occurs when the time
  -- specified has already past
  --*D*   store 0 3 prior_time_warning \
  --*D*   :short = "The specified time has already past.
  --*D*Command is executed immediately."
  --
  -- End of Header


  type program_record is record
    -- times in this record are all in
    -- system_time_units to be used by Timed_request
```

```
    command:     System_Defs.text(80);
      -- command to be run with arguments
    stu_start:   System_Defs.system_time_units;
      -- initial request
    stu_period:  System_Defs.system_time_units;
      -- interval between execution (optional argument)
    stu_until:   System_Defs.system_time_units;
      -- upper time limit on command run more than once
    count:       integer;
      -- number of times job will run
  end record;



  function Parse_time(
      time:       System_Defs.string_list;
            -- time from command line
      from_when: Timed_Requests_Mgt.from_when_type)
            -- specifies time to be relative to now
            -- or absolute
    return System_Defs.system_time_units;
            -- time in form usable for
            -- Timed_Request.Enter_request
  --
  -- Function:
  --   Parses the time argument on the command line and
  --   converts to system_time_units. The time
  --   specification is divided into two strings, the
  --   first being mandatory specifying hours and
  --   minutes and optionally seconds and hundredths of
  --   seconds.  The second string is optional and
  --   specifies the day of month and optionally the
  --   month and year.
  --
  -- Exceptions:
  --   time_format_error - raised when the hour string list
  --                       input for the timing
  --                       parameters is incorrect.
  --
  --   day_format_error - raised when the day string list
  --                      input for the timing parameters
  --                      is incorrect.


  procedure Create_waiting_process(
      invocation_record: program_record);
  --
  -- Function:
  --   Creates a new session, job and process to wait
  --   for specified time to execute.
  --



  procedure Wait_program(
      param_buffer: System.address;
      param_length: System.ordinal);
    pragma subprogram_value(Process_Mgt.initial_proc,
        Wait_program);
  --
  -- Function:
  --   Created in a new session and job. Process issues
  --   a timed request and waits on the locked semaphore
  --   for specified time to execute program passed in
  --   as a parameter.  If the command is specified more
  --   than once it will loop, issue another timing
  --   request and reset the semaphore and wait.
  --
  --
end At_Support_Ex;
```

X-A.6.3 At_Support_Ex Package Body

```
with Command_Execution,
     Directory_Mgt,
     Job_Admin,           -- trusted
     Job_Mgt,
     Job_Types,
     Long_Integer_Defs,
     Incident_Defs,
     Message_Services,
     Message_Stack_Mgt,
     Semaphore_Mgt,
     Session_Mgt,
     Session_Types,
     String_List_Mgt,
     System,
     System_Defs,
     Text_IO,
     Text_Mgt,
     Timed_Requests_Mgt,
     Timing_String_Conversions,
     Timing_Conversions;

package body At_Support_Ex is
  --
  -- Logic:
  --   Supports at command by parsing time specification and creating
  --   new session, job and process that will wait for timing requests
  --   to invoke the waiting process.
  --
  -- History:
  --   04-05-88, Ed Sassone, creation date
  --   05-20-88, Ed Sassone, working version
  --
  -- End of Header

---------------------------------------------------------
--                    PARSE_TIME                       --
---------------------------------------------------------
  function Parse_time(
      time:      System_Defs.string_list;
      from_when: Timed_Requests_Mgt.from_when_type)
    return System_Defs.system_time_units
  --
  -- Logic:
  --   This function first parses the mandatory string
  --   containing hours, minutes, seconds, hundreths and
  --   then it parses the second optional string
  --   containing month day and year.  For each string
  --   it counts the number and position of the
  --   separator.  For the first string that is the ':'
  --   and the '.' if hundreths are specified.
  --   For the second string it is the '/'.  Based on the
  --   separator positions, substrings representing the
  --   individual time elements are copied into the
  --   appropriate fields of string_time.
  --
  is


    use Timed_Requests_Mgt;
      -- needed in "if from_when = system_epoch statement"


    dum_text:           constant System_Defs.text(11) :=
        (11, 11, (others => ' '));
      -- used for the following initialization only:

    string_time:        Timing_String_Conversions.string_time :=
        ("0000", "   ", "00", "00", "00", "00", "00", dum_text,
         "   ", "   ");
      -- specified time values are copied into fields if
      -- absolute time is used value is preloaded with
      -- current time.  Fields specified are overwritten

    string_interval:    Timing_String_Conversions.string_interval;
```

```
      -- used for period (relative time)

    hour_time:          System_Defs.text(Incident_Defs.txt_length);
      -- used for hh:mm:ss.dd field

    day_time:           System_Defs.text(Incident_Defs.txt_length);
      -- used for MM/DD/YYYY field

    separators:         array (1 .. 2) of
        System_Defs.text_length;
      -- array of positions of separators

    number_separators:  integer := 0;
      -- hold the number of separators in the field

    month:              string (1  ..  2) :=  "00";
      -- used in place of string_time.month because
      -- string_time.month is Jan..Dec and specified
      -- month is 1..12

    package Int_IO is new Text_IO.Integer_IO(integer);
      -- needed for conversions from string to numeric
      -- month

  begin

    -- initialize string_time record

    if from_when = system_epoch then
      -- absolute time for current day
      string_time := Timing_String_Conversions.
          Convert_numeric_time_to_string(
              num_time => Timing_Conversions.
                  Convert_stu_to_numeric_time(
                      stu  => Timed_Requests_Mgt.
                          Get_time));  -- current time

      -- default if not specified
      string_time.minute := "00";
      string_time.second := "00";
      string_time.hundredth := "00";
    end if;

    -- *** PARSE MANDATORY HOUR STRING ***

    String_List_Mgt.Get_element(
        from    => time,
        el_pos  => 1,
        element => hour_time);

    -- find positions and number of ":"
    number_separators := 0;
    separators := (others => 0);
    for pos in 1 .. hour_time.length
      loop
      if hour_time.value(pos) = ':' then
        number_separators := number_separators + 1;
        -- no more than 2 ":" allowed
        if number_separators > 2 then
          RAISE time_format_error;
        end if;
        separators(number_separators) := pos;
      -- if non-digit or not the other separator
      elsif (hour_time.value(pos) < '0'  or
          hour_time.value(pos) > '9') and
          hour_time.value(pos) /= '.' then
            RAISE time_format_error;
      end if;
    end loop;

    case number_separators is
      when 0 =>
        if hour_time.length > 2 then
          RAISE time_format_error;
        end if;
        string_time.hour := hour_time.value;
      when 1 =>
```

```
     if separators(1) /= 3 then
       RAISE time_format_error;
     end if;
     string_time.hour := hour_time.value(1 .. 2);
     string_time.minute := hour_time.value(4 .. 5);
   when 2 =>
     if separators(1) /= 3 or separators(2) /= 6 then
       RAISE time_format_error;
     end if;
     string_time.hour := hour_time.value(1 .. 2);
     string_time.minute := hour_time.value(4 .. 5);
     string_time.second := hour_time.value(7 .. 8);

     -- do hundredths if specified
     declare
       pos: integer := Text_Mgt.Locate('.', hour_time);
     begin
       case  pos is
         when 0 =>
           null;
         when 9 =>
           string_time.hundredth := hour_time.value
               (pos + 1 .. pos + 2);
         when others =>
             RAISE time_format_error;
       end case;
     end;          -- declare
   when others =>
     RAISE time_format_error;
 end case;

-- *** PARSE OPTIONAL DAY STRING ***

 if time.count = 2 then
    String_List_Mgt.Get_element(
       from    => time,
       el_pos  => 2,
       element => day_time);

   -- find positions of "/"
   number_separators := 0;
   separators := (others => 0);
   for pos in 1 .. day_time.length
   loop
     if day_time.value(pos) = '/' then
       number_separators := number_separators + 1;
       -- no more than 2 "/" allowed
       if number_separators > 2 then
         RAISE day_format_error;
       end if;
       separators(number_separators) := pos;
     -- digits only if not a valid separator
     elsif day_time.value(pos) < '0' or
         day_time.value(pos) > '9' then
       RAISE day_format_error;
     end if;
   end loop;

   case number_separators is
     when 0 =>
       -- day of month only
       string_time.day := day_time.value;

     when 1 =>
       -- month and day
       if separators(1) /= 3 then
         RAISE day_format_error;
       end if;
       month := day_time.value(1 .. 2);
       string_time.day := day_time.value(4 .. 5);

     when 2 =>
       -- month, day and year
       if separators(1) /= 3 or separators(2) /= 6 then
         RAISE day_format_error;
       end if;
       month := day_time.value(1 .. 2);
```

```
          string_time.day := day_time.value(4 .. 5);
          string_time.year := day_time.value(7 .. 10);

        when others =>
          RAISE day_format_error;
      end case;


      -- convert 1..12 month to Jan..Dec month
      declare

        month_tmp:              integer;
          -- temporary variable for month conversion

        length:         positive;
          -- dummy variable for month conversion

      begin

          Int_IO.get(           -- convert string to ordinal
              from => month,
              item => month_tmp,
              last => length);

          case month_tmp is
            when 0 =>
              null;  --blank initial string
            when 1 =>
              string_time.month := "Jan";
            when 2 =>
              string_time.month := "Feb";
            when 3 =>
              string_time.month := "Mar";
            when 4 =>
              string_time.month := "Apr";
            when 5 =>
              string_time.month := "May";
            when 6 =>
              string_time.month := "Jun";
            when 7 =>
              string_time.month := "Jul";
            when 8 =>
              string_time.month := "Aug";
            when 9 =>
              string_time.month := "Sep";
            when 10 =>
              string_time.month := "Oct";
            when 11 =>
              string_time.month := "Nov";
            when 12 =>
              string_time.month := "Dec";
            when others =>
              RAISE day_format_error;
          end case;
      end;                      -- declare

    end if;         -- if time.count = 2

    -- range checking goes here

    if from_when = system_epoch then
      -- absolute time
      return Timing_Conversions.Convert_numeric_time_to_stu(
          num_time => Timing_String_Conversions.
              Convert_string_time_to_numeric(
                  str_time => string_time));
    else
      -- relative time
      -- initialize to zero
      string_interval := Timing_String_Conversions.
          Convert_numeric_interval_to_string(
              num_interval => Timing_Conversions.
                  Convert_stu_to_numeric_interval(
                      stu => System_Defs.null_time));

      string_interval.sign := ' ';
      string_interval.days(7 .. 8) := string_time.day;
```

```
      string_interval.hours(11 .. 12) := string_time.hour;
      string_interval.minutes(11 .. 12) := string_time.minute;
      string_interval.seconds(11 .. 12) := string_time.second;
      string_interval.hundredths(11 .. 12) := string_time.hundredth;

      return Timing_Conversions.Convert_numeric_interval_to_stu(
          num_interval => Timing_String_Conversions.
              Convert_string_interval_to_numeric(
                  str_interval => string_interval));
    end if;

    exception

      when time_format_error =>
        Message_Services.Write_msg(
            msg_id => time_format_error_code,
            param1 => Incident_Defs.message_parameter'(
                typ     => Incident_Defs.txt,
                len     => Incident_Defs.txt_length,
                txt_val => hour_time));
      RAISE;

      when day_format_error =>
        Message_Services.Write_msg(
            msg_id => day_format_error_code,
            param1 => Incident_Defs.message_parameter'(
                typ     => Incident_Defs.txt,
                len     => Incident_Defs.txt_length,
                txt_val => day_time));
      RAISE;

  end Parse_time;


---------------------------------------------------------
--               CREATE_WAITING_PROCESS                --
---------------------------------------------------------
  procedure Create_waiting_process(
      invocation_record: program_record)
  is
    --
    -- Logic:
    --  Creates a new session, then a job in that session,
    --  and then the waiting process from that job.
    --

    new_name:        constant System_Defs.text(13) :=
        (13, 13, "timed request");

    job_info:        Job_mgt.job_info(80);
      -- SSO field used for creating new session

    new_job_AD:      Job_Types.job_AD;

    program_length:  System.ordinal := System.ordinal(
        invocation_record'size / System.storage_unit);

  begin

    -- retrieves SSO for new session
    Job_Mgt.Get_job_info(
        info => job_info);

    new_job_AD := Job_Admin.Invoke_job(
        init_proc    => Wait_program'subprogram_value,
        param_buffer => invocation_record'address,
        param_length => program_length,
        text         => new_name,
        session      => Session_Mgt.create_session(
            SSO          => job_info.SSO,
            session_name => new_name));

  end Create_waiting_process;


---------------------------------------------------------
--                   WAIT_PROGRAM                      --
```

```
---------------------------------------------------------
  procedure Wait_program(
      param_buffer: System.Address;
      param_length: System.ordinal)

  is


    use Long_Integer_Defs;
      -- for system_time_units


    program_rec :    program_record;
    FOR program_rec USE AT param_buffer;

    command_job_AD:  Job_Types.job_Ad;

    req_index:       Timed_Requests_Mgt.request_index;

    wait:            Semaphore_Mgt.semaphore_AD :=
        Semaphore_Mgt.Create_semaphore(
            initial_count => 0);
       -- create semaphore in locked state
       -- blocks job until time specified

  begin

    -- period must be non-null for
    -- Timed_Requests_Mgt.Get_next_activation
    if program_rec.stu_period = System_Defs.null_time then
      program_rec.stu_period := System_Defs.stu_per_min;
    end if;

    -- Loop until count is expired or "until" time is
    -- expired, whichever is first.  Count and until both
    -- have defaults of max_int.  If period was not specified
    -- the loop count was set to one by the driver
    while program_rec.stu_until >= program_rec.stu_start
        and program_rec.count > 0
    loop
      req_index :=
          Timed_Requests_Mgt.Enter_request(
              req_info => Timed_Requests_Mgt.request_info(
                  Timed_Requests_Mgt.semaphore_signal)'(
                     kind        => Timed_Requests_Mgt.semaphore_signal,
                     wakeup_time => program_rec.stu_start,
                     from_when   => Timed_Requests_Mgt.system_epoch,
                     semaphore   => wait));

       -- wait until Timed_Requests unlocks semaphore
       -- NOTE: there is about a 3 second delay before the
       -- command is actually run
      Semaphore_Mgt.P(semaphore => wait);

      command_job_AD :=
          Command_Execution.Run_program_or_script(
          command => program_rec.command);

      program_rec.count := program_rec.count - 1;

      -- NOTE1: This is an expensive call that should only be
      -- used when slippage cannot be tolerated.
      -- NOTE2: The call should be placed after command invocation.
      Timed_Requests_Mgt.Get_next_activation(
          period          => program_rec.stu_period, -- this cannot be null
          next_activation => program_rec.stu_start);

    end loop;

  end Wait_program;

end At_Support_Ex;
```

X-A.6.4 `Compiler_Ex` Package Specification

```
with Device_Defs;

package Compiler_Ex is
  --
  -- Function:
  --   Supplies the procedural interface a Pascal
  --   compiler.
  --
  --   This interface can be used to write the
  --   compiler invocation script. End of Header
  --
  -- History:
  --   08-10-87, Paul Schwabe: initial revision.
  --   12-02-87, Paul Schwabe: revision.
  pragma external;

  procedure Compile_pascal(
    source_code:   Device_Defs.opened_device;
      -- Opened on source code input file, with read
      -- rights.
    machine_code:  Device_Defs.opened_device;
      -- Opened on machine code output file, with read
      -- and write rights.
    listing:       Device_Defs.opened_device);
      -- Opened on listing output file, with write
      -- rights.
    --
    -- Function:
    --   Compiles a Pascal program.
    --
    --   Relies on the caller to handle user
    --   interaction.

end Compiler_Ex;
```

X-A.6.5 Compiler_Ex Package Body

```
with Byte_Stream_AM,
     Device_Defs,
     Event_Mgt,
     Pipe_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     System;

package body Compiler_Ex is
  --
  -- Logic:
  --    Speeds up a Pascal compiler by dividing parsing
  --    and code generation between two processes
  --    connected by a pipe.
  --
  --    "Parse" and "Code_gen" are the initial
  --    procedures of the two child processes.
  --
  -- History:
  --    11-24-87, Paul Schwabe:  Initial version.
  --    11-25-87, Gary Taylor :  Added tagged comment lines.
  --
  -- End of Header


type connection_record is record
  --    A "connection_record" contains the I/O
  --    connections used by the two child processes.
  --    The entire record is passed to both children.

  source_code:   Device_Defs.opened_device;
    -- input file
  machine_code:  Device_Defs.opened_device;
    -- output file
  listing:       Device_Defs.opened_device;
    -- output file
  parse_out:     Device_Defs.opened_device;
    -- output to pipe
  code_gen_in:   Device_Defs.opened_device;
    -- input from pipe
end record;


procedure Parse(
    param_buffer:  System.address;
      -- Address of connection record.
    param_length:  System.ordinal)
      -- Not used in this procedure, but required for
      -- process's initial procedure.
  --
  -- Logic:
  --    Do Pascal parsing using the I/O connections
  --    specified in the "conn_rec" parameter record.
is
  conn_rec:  connection_record;  -- Record containing
                                 -- parameters.
  FOR conn_rec USE AT param_buffer;
begin
  -- Code to parse "conn_rec.source_code" and write
  -- parsed stream to "conn_rec.parse_out" and listing
  -- to "conn_rec.listing" goes here.
  null;
end Parse;
pragma subprogram_value(Process_Mgt.Initial_proc, Parse);


procedure Code_gen(
    param_buffer:  System.address;
      -- Address of connection record.
    param_length:  System.ordinal)
      -- Not used but required for process's initial
      -- procedure.
  --
  -- Logic:
```

```
  --   Do Pascal code generation using the I/O
  --   connections specified in the "conn_rec"
  --   parameter record.
is
  conn_rec:  connection_record;
    -- Record containing parameters.
  FOR conn_rec USE AT param_buffer;
begin
  -- Code to read "conn_rec.code_gen_in", write
  -- compiled code to "conn_rec.machine_code", and add
  -- any needed messages to "cr.listing" goes here.
  null;
end Code_gen;
  pragma subprogram_value(
      Process_Mgt.Initial_proc,
      Code_gen);


procedure Compile_pascal(
    source_code:   Device_Defs.opened_device;
    machine_code:  Device_Defs.opened_device;
    listing:       Device_Defs.opened_device)
  --
  -- Logic:
  --
  --   1. Create a pipe.
  --
  --   2. Create a record specifying all I/O
  --      connections for child processes.  Open both
  --      ends of the pipe to create the pipe
  --      connections needed.
  --
  --   3. Get an AD for this process from process
  --      globals.
  --
  -- 4. Spawn the parsing process.  The parameter
  --    buffer address is the connection record's
  --    address.  The termination action signals the
  --    "user_1" event to this process.
  --
  --   5. Spawn the code generation process.  The
  --      parameter buffer address is the connection
  --      record's address. The termination action
  --      signals the "user_2" event to this process.
  --
  --   6. Wait for both the "user_1" and "user_2"
  --      events to be signalled indicating that both
  --      child processes have terminated.
  --
  --   7. Deallocate both child processes.
  --
  -- Notes:
  --   No check is made for abnormal termination of
  --   the child processes.
  --
  --   Would like to deallocate pipe when done with it
  --   but "Pipe_Mgt" does not provide a "Deallocate"
  --   call.
is
  compiler_pipe:  Pipe_Mgt.pipe_AD;
    -- Pipe that connects "Parse" and "Code_gen"
    -- processes.

  conn_rec:  connection_record;
    -- Record referencing all I/O connections used by
    -- the child processes.

  this_process_untyped:  System.untyped_word;
    -- Process executing call to "Compile_pascal",
    -- as an "untyped_word".

  parse_process:  Process_Mgt_Types.process_AD;
    -- Process executing "Parse".

  code_gen_process:  Process_Mgt_Types.process_AD;
    -- Process executing "Code_gen".

```

```
  term_events:  Event_Mgt.action_record_list(2);
    -- Array that receives termination events of the
    -- two child processes.

begin
  compiler_pipe := Pipe_Mgt.Create_pipe;

  conn_rec := (
      source_code  => source_code,
      machine_code => machine_code,
      listing      => listing,
      parse_out    => Byte_Stream_AM.Ops.Open(
          Pipe_Mgt.Convert_pipe_to_device(
              compiler_pipe),
          Device_Defs.output),
      code_gen_in   => Byte_Stream_AM.Ops.Open(
          Pipe_Mgt.Convert_pipe_to_device(
              compiler_pipe),
          Device_Defs.input));

  this_process_untyped :=
      Process_Mgt.Get_process_globals_entry(
          Process_Mgt_Types.process);

  parse_process := Process_Mgt.Spawn_process(
      init_proc    => Parse'subprogram_value,
      param_buffer => conn_rec'address,
      term_action  => (
          event =>       Event_Mgt.user_1,
          message =>     System.null_address,
          destination => this_process_untyped));

  code_gen_process := Process_Mgt.Spawn_process(
      init_proc    => Code_gen'subprogram_value,
      param_buffer => conn_rec'address,
      term_action  => (
          event =>       Event_Mgt.user_2,
          message =>     System.null_address,
          destination => this_process_untyped));

  Event_Mgt.Wait_for_all(
      events =>
          (Event_Mgt.user_1 .. Event_Mgt.user_2 =>
               true,
           others => false),
      action_list => term_events);

  -- These process are terminated so
  -- "Deallocate" should work.
  Process_Mgt.Deallocate(parse_process);
  Process_Mgt.Deallocate(code_gen_process);

end Compile_pascal;


end Compiler_Ex;
```

X-A.6.6 `Conversion_Support_Ex` Package Specification

```
with Attribute_Mgt,
     Authority_List_Mgt,
     Data_Definition_Mgt,
     Device_Defs,
     Directory_Mgt,
     Event_Mgt,
     File_Defs,
     Identification_Mgt,
     Identification_Mgt,
     Job_Types,
     Name_Space_Mgt,
     Object_Mgt,
     Object_Mgt,
     Passive_Store_Mgt,
     Pipe_Mgt,
     Process_Mgt_Types,
     Session_Types,
     System_Defs,
     System,
     Unchecked_conversion;

package Conversion_Support_Ex is
  --
  -- Function:
  --   Provides commonly needed compile-time type
  --   conversions for OS access types.
  --
  --   Some OS calls can operate on many different
  --   object types. Such calls require or return
  --   values of type "System.untyped_word", used to
  --   hold any AD.  If your application uses ADs with
  --   more specific types, you must convert those
  --   types to and from "System.untyped_word". For
  --   example, to store an AD to a Type Definition
  --   Object in a directory, you must convert from
  --   the type "Object_Mgt.TDO_AD" to
  --   "System.untyped_word".
  --
  --   All the conversion routines in this package are
  --   instantiations of the "Unchecked_conversion"
  --   generic Ada function.  Calls to the conversion
  --   routines are processed at compile-time, and
  --   have no runtime cost.
  --
  --   There are a few conversions that don't require
  --   using a conversion routine.  For example,
  --   "Device_Defs.device" is a subtype of
  --   "System.untyped_word".  This package still
  --   provides the expected conversion routines--they
  --   have no runtime cost, and by using them you do
  --   not have to remember which types don't require
  --   conversion.
  --
  --   The conversion function names have the form
  --   "X_from_Y" where "X" indicates the result type
  --   and "Y" indicates the source type.
  --
  -- History:
  --   06-03-87, Martin L. Buchanan:  Initial version.
  --   06-09-87, Paul Schwabe: Added full set of
  --   unchecked_conversions.
  --   11-23-87, Paul Schwabe: Fixed line sizes.
  --
  -- End of Header
  pragma external;

  function Attribute_ID_from_untyped is new
     Unchecked_conversion(
        source => System.untyped_word,
        target => Attribute_Mgt.attribute_ID_AD);


  function Untyped_from_attribute_ID is new
      Unchecked_conversion(
```

```
            source => Attribute_Mgt.attribute_ID_AD,
            target => System.untyped_word);


    function Authority_list_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Authority_List_Mgt.
                authority_list_AD);


    function Untyped_from_authority_list is new
        Unchecked_conversion(
            source => Authority_List_Mgt.
                authority_list_AD,
            target => System.untyped_word);


    function DDef_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Data_Definition_Mgt.DDef_AD);


    function Untyped_from_DDef is new
        Unchecked_conversion(
            source => Data_Definition_Mgt.DDef_AD,
            target => System.untyped_word);


    function Device_from_untyped is new
        Unchecked_conversion(
            source => Device_Defs.device,
            target => Authority_List_Mgt.
                authority_list_AD);


    function Untyped_from_device is new
        Unchecked_conversion(
            source => Authority_List_Mgt.
                authority_list_AD,
            target => Device_Defs.device);


    function Opened_device_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Device_Defs.opened_device);


    function Untyped_from_opened_device is new
        Unchecked_conversion(
            source => Device_Defs.opened_device,
            target => System.untyped_word);


    function Directory_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Directory_Mgt.directory_AD);


    function Untyped_from_directory is new
        Unchecked_conversion(
            source => Directory_Mgt.directory_AD,
            target => System.untyped_word);


    function Event_cluster_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Event_Mgt.event_cluster_AD);


    function Untyped_from_event_cluster is new
        Unchecked_conversion(
            source => Event_Mgt.event_cluster_AD,
```

```
        target => System.untyped_word);


function File_from_untyped is new
    Unchecked_conversion(
        source => System.untyped_word,
        target => File_Defs.file_AD);


function Untyped_from_file is new
    Unchecked_conversion(
        source => File_Defs.file_AD,
        target => System.untyped_word);


function ID_from_untyped is new
    Unchecked_conversion(
        source => System.untyped_word,
        target => Identification_Mgt.ID_AD);


function Untyped_from_ID is new
    Unchecked_conversion(
        source => Identification_Mgt.ID_AD,
        target => System.untyped_word);


function ID_list_from_untyped is new
    Unchecked_conversion(
        source => System.untyped_word,
        target => Identification_Mgt.ID_list_AD);


function Untyped_from_ID_list is new
    Unchecked_conversion(
        source => Identification_Mgt.ID_list_AD,
        target => System.untyped_word);


function Job_from_untyped is new
    Unchecked_conversion(
        source => System.untyped_word,
        target => Job_Types.job_AD);


function Untyped_from_job is new
    Unchecked_conversion(
        source => Job_Types.job_AD,
        target => System.untyped_word);


function Name_space_from_untyped is new
    Unchecked_conversion(
        source => System.untyped_word,
        target => Name_Space_Mgt.name_space_AD);


function Untyped_from_name_space is new
    Unchecked_conversion(
        source => Name_Space_Mgt.name_space_AD,
        target => System.untyped_word);


function SRO_from_untyped is new
    Unchecked_conversion(
        source => System.untyped_word,
        target => Object_Mgt.SRO_AD);


function Untyped_from_SRO is new
    Unchecked_conversion(
        source => Object_Mgt.SRO_AD,
        target => System.untyped_word);


function TDO_from_untyped is new
    Unchecked_conversion(
```

```
            source => System.untyped_word,
            target => Object_Mgt.TDO_AD);


   function Untyped_from_TDO is new
       Unchecked_conversion(
            source => Object_Mgt.TDO_AD,
            target => System.untyped_word);


   function PSM_attributes_from_untyped is new
       Unchecked_conversion(
            source => System.untyped_word,
            target => Passive_Store_Mgt.
                PSM_attributes_AD);


   function Untyped_from_PSM_attributes is new
       Unchecked_conversion(
            source => Passive_Store_Mgt.
                PSM_attributes_AD,
            target => System.untyped_word);


   function Pipe_from_untyped is new
       Unchecked_conversion(
            source => System.untyped_word,
            target => Pipe_Mgt.pipe_AD);


   function Untyped_from_pipe is new
       Unchecked_conversion(
            source => Pipe_Mgt.pipe_AD,
            target => System.untyped_word);


   function Process_from_untyped is new
       Unchecked_conversion(
            source => System.untyped_word,
            target => Process_Mgt_Types.process_AD);


   function Untyped_from_process is new
       Unchecked_conversion(
            source => Process_Mgt_Types.process_AD,
            target => System.untyped_word);


   function Session_from_untyped is new
       Unchecked_conversion(
            source => System.untyped_word,
            target => Session_Types.session_AD);


   function Untyped_from_session is new
       Unchecked_conversion(
            source => Session_Types.session_AD,
            target => System.untyped_word);


   function Text_from_untyped is new
       Unchecked_conversion(
            source => System.untyped_word,
            target => System_Defs.text_AD);


   function Untyped_from_text is new
       Unchecked_conversion(
            source => System_Defs.text_AD,
            target => System.untyped_word);


end Conversion_Support_Ex;
```

X-A.6.7 Memory_ex Procedure

```
with Object_Mgt,
     Long_Integer_Defs,
     SRO_Mgt,
     System_Defs;

procedure Memory_ex
  --
  -- Function:
  --   Provide examples of several memory management
  --   programming techniques.

is
  -- Declare a record for a job's memory
  -- information:
  --
  job_memory_info:  SRO_Mgt.SRO_information;
begin

  -- Get current memory information for the calling
  -- job:
  --
  job_memory_info := SRO_Mgt.Read_SRO_information;


  -- Shrink the calling process's stack to the
  -- size currently used.  The stack can still
  -- grow and will be expanded as needed.
  --
  Object_Mgt.Trim_stack;


  -- Force a local garbage collection run to start
  -- immediately in the calling job:
  --
  SRO_Mgt.Start_GCOL;


  -- Configure a local garbage collection daemon
  -- to run in the calling job when it has used
  -- 50% of its storage claim OR 50% of its object
  -- table page claim, AND at least 5 minutes
  -- has elapsed since a previous local GCOL run
  -- in the job.
  --
  SRO_Mgt.Start_GCOL(
      storage_claim_percent => 50,
      OTP_claim_percent     => 50,
      minimum_delay         =>
          Long_Integer_Defs."*"(
          Long_Integer_Defs.long_integer'(0, 5),
          System_Defs.stu_per_min));


  -- Kill any local garbage collection daemon in
  -- the calling job.  (Does nothing if there
  -- is no daemon.)
  --
  SRO_Mgt.Start_GCOL(0, 0, Long_Integer_Defs.max_int);

end Memory_ex;
```

X-A.6.8 `Process_Globals_Support_Ex` Package Specification

```
with Authority_List_Mgt,
     Device_Defs,
     Directory_Mgt,
     Identification_Mgt,
     Job_Types,
     Name_Space_Mgt,
     Process_Mgt_Types,
     Session_Types,
     System_Defs;

package Process_Globals_Support_Ex is

  -- Function:
  --   Provide calls to get and set commonly used
  --   process globals entries, for the calling
  --   process.
  --
  --   See "Process_Mgt_Types" for descriptions of all
  --   process globals entries.
  --
  --
  --   << What You Get with This Package >>
  --
  --   There are three advantages to using this
  --   package, as compared to using the "Process_Mgt"
  --   calls to get and set process globals:
  --
  --   1. The underlying calls require or return
  --      untyped words.  You must instantiate
  --      "Unchecked_conversion" to convert to and from
  --      the types you actually need, such as
  --      "Device_Defs.opened_device".
  --
  --   2. You don't have to supply a value of type
  --      "Process_Mgt_Types.process_globals_entry" that
  --      specifies the process globals *slot* you are
  --      manipulating.
  --
  --   3. The underlying calls can be used to stuff
  --      garbage into process globals entries and later
  --      return that garbage.  The calls in this
  --      package do reasonable checks on type, rights,
  --      and object state for the modifiable process
  --      globals entries. Such checks aren't needed for
  --      the non-modifiable entries, assigned by
  --      the OS.
  --
  --
  --   << What You Don't Get with This Package >>
  --
  --   This package does not support assigning or
  --   retrieving null values for the modifiable
  --   process globals entries. You can assign and
  --   retrieve null values for these entries using
  --   "Process_Mgt" calls.
  --
  --   This package does not support getting or
  --   setting another process's globals.  You can
  --   access another process's globals by using
  --   "Process_Mgt" or "Process_Admin" calls.
  --
  --   This package does not support setting any
  --   process globals entries that can only be set by
  --   an administrative interface, such as
  --   "Process_Admin".
  --
  --   This package is selective, and does not provide
  --   calls to get or set every publicly accessible
  --   entry.
  --
  -- Exceptions:
  --   user_dialog_not_interactive
  --
  --
```

```
-- History:
--   06-03-87, Martin L. Buchanan:  Initial version.
--   11-23-87, Paul Schwabe: Updated spec.
--
-- End of Header
pragma external;

function Get_standard_input
  return Device_Defs.opened_device;
    -- The calling process's standard
    -- input opened device,
    -- open and with read rights.
  --
  -- Function:
  --   Returns the calling process's standard input.
  --
  -- Exceptions:
  --   Device_Defs.device_not_open -
  --     The opened device has been closed.


procedure Set_standard_input(
  opened_dev:  Device_Defs.opened_device);
      -- Opened device, open and with read rights.
  --
  -- Function:
  --   Assigns the calling process's standard input.
  --
  -- Exceptions:
  --   Device_Defs.device_not_open -
  --     The opened device has been closed.


function Get_standard_output
  return Device_Defs.opened_device;
    -- The calling process's standard
    -- output opened device,
    -- open and with write rights.
  --
  -- Function:
  --   Returns the calling process's standard output.
  --
  -- Exceptions:
  --   Device_Defs.device_not_open -
  --     The opened device has been closed.


procedure Set_standard_output(
  opened_dev:  Device_Defs.opened_device);
      -- Opened device, open and with write rights.
  --
  -- Function:
  --   Assigns the calling process's standard output.
  --
  -- Exceptions:
  --   Device_Defs.device_not_open -
  --     The opened device has been closed.


function Get_standard_message
  return Device_Defs.opened_device;
    -- The calling process's standard
    -- message opened device,
    -- open and with write rights.
  --
  -- Function:
  --   Returns the calling process's standard message
  --   opened device.
  --
  -- Exceptions:
  --   Device_Defs.device_not_open -
  --     The opened device has been closed.


procedure Set_standard_message(
  opened_dev:  Device_Defs.opened_device);
      -- Opened device, open and with write rights.
```

```
    --
    -- Function:
    --   Assigns the calling process's standard
    --   message opened device.
    --
    -- Exceptions:
    --   Device_Defs.device_not_open -
    --     The opened device has been closed.


  function Get_user_dialog
    return Device_Defs.opened_device;
      -- The calling process's user
      -- dialog opened device, open, with the
      -- "is_interactive" flag set in the
      -- underlying device's information record,
      -- and with both read and write rights.
    --
    -- Function:
    --   Returns the calling process's
    --   user dialog opened device.
    --
    -- Exceptions:
    --   Device_Defs.device_not_open -
    --     The opened device has been closed.


  procedure Set_user_dialog(
    opened_dev:  Device_Defs.opened_device);
      --   An opened device that is open, with the
      --   "is_interactive" flag set in the underlying
      --   device's information record, and with both
      --   read and write rights.
    --
    -- Function:
    --   Assigns the calling process's user dialog
    --   opened device.
    --
    -- Exceptions:
    --   Device_Defs.device_not_open -
    --     The opened device has been closed.


  function Get_home_directory
    return Directory_Mgt.directory_AD;
      -- The calling process's home directory.
    --
    -- Function:
    --   Returns the calling process's home directory.
    --
    -- Notes:
    --   Setting a process's home directory is an
    --   administrative operation.


  function Get_current_directory
    return Directory_Mgt.directory_AD;
      -- The calling process's current directory.
    --
    -- Function:
    --   Returns the calling process's current
    --   directory.


  procedure Set_current_directory(
    dir:  Directory_Mgt.directory_AD);
        -- Any directory.
    --
    -- Function:
    --   Assigns the calling process's current
    --   directory.


  function Get_authority_list
    return Authority_List_Mgt.authority_list_AD;
      -- The calling process's authority list.
    --
```

```
    -- Function:
    --   Returns the calling process's authority list.


  procedure Set_authority_list(
  auth:  Authority_List_Mgt.authority_list_AD);
      -- Any authority list.
    --
    -- Function:
    --   Assigns the calling process's default
    --   authority list.


  function Get_ID_list
    return Identification_Mgt.ID_list_AD;
      -- The calling process's ID list.
    --
    -- Function:
    --   Returns the calling process's ID list.
    --
    -- Notes:
    --   Setting a process's ID list is an
    --   administrative operation.


  function Get_command_name_space
    return Name_Space_Mgt.name_space_AD;
      -- The calling process's command name space.
    --
    -- Function:
    --   Returns the calling process's command name
    --   space.


  procedure Set_command_name_space(
    ns:  Name_Space_Mgt.name_space_AD);
        -- Any name space.
    --
    -- Function:
    --   Assigns the calling process's command name
    --   space.


  function This_process
    return Process_Mgt_Types.process_AD;
      -- The calling process, with control rights.
    --
    -- Function:
    --   Returns the calling process.


  function Get_parent_process
    return Process_Mgt_Types.process_AD;
      --   Parent process of the calling process, with
      --   control rights.  Null if the calling
      --   process is the initial process of its job.
    --
    -- Function:
    --   Returns the calling process's parent process,
    --   if any.


  function This_job
    return Job_Types.job_AD;
      --   Job that contains the calling process, with
      --   list and control rights.
    --
    -- Function:
    --   Returns the calling job.


  function This_session
    return Session_Types.session_AD;
      --   Session that contains the calling job, with
      --   list and control rights.
    --
    -- Function:
```

```
      --   Returns the caller's session.


    function Get_process_name
      return System_Defs.text_AD;
        --   AD to text record containing the calling
        --   process's name.
      --
      -- Function:
      --   Returns the calling process's symbolic name.
      --
      --   The symbolic name may be a null text record.


    procedure Set_process_name(
      name:  System_Defs.text);
          --   A text record containing a name for the
          --   process.  The text record must be valid,
          --   with a "length" field less than or equal
          --   to its "max_length" field.
      --
      -- Function:
      --   Assigns the calling process's symbolic name.
      --
      -- Exceptions:
      --   System_Exceptions.bad_parameter
      --

  end Process_Globals_Support_Ex;
```

X-A.6.9 Process_Globals_Support_Ex Package Body

```
with Access_Mgt,
     Authority_List_Mgt,
     Byte_Stream_AM,
     Device_Defs,
     Directory_Mgt,
     Identification_Mgt,
     Job_Mgt,
     Job_Types,
     Name_Space_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     Session_Mgt,
     Session_Types,
     System_Defs,
     System_Exceptions,
     System;

package body Process_Globals_Support_Ex is

  -- Function:
  --   Provide calls to get and set commonly used
  --   process globals entries, for the calling
  --   process.
  --
  -- History:
  --   06-10-87, Paul Schwabe:  Initial version.
  --   11-24-87, Paul Schwabe:  Updated version.
  --   11-25-87, Gary Taylor :  Added tagged comment lines.
  --
  -- End of Header


  function Get_standard_input
    return Device_Defs.opened_device
    --
    -- Logic:
    --   1. Get the process globals entry.
    --   2. Check that the standard input is open,
    --      which implicitly checks that its an opened
    --      device.
    --   3. Check that the standard input has
    --      read rights.
    --   4. Return the standard input.
  is
    stdin:          Device_Defs.opened_device;
    stdin_untyped:  System.untyped_word;
      FOR stdin_untyped USE AT stdin'address;
  begin
    stdin_untyped := Process_Mgt.
        Get_process_globals_entry(
            Process_Mgt_Types.standard_input);

    if not Byte_Stream_AM.Ops.Is_open(stdin) then
      RAISE Device_Defs.device_not_open;

    elsif not Access_Mgt.Permits(
       AD     =>  stdin_untyped,
       rights =>  Device_Defs.read_rights) then
      RAISE System_Exceptions.insufficient_type_rights;

    else
      RETURN stdin;

    end if;
  end Get_standard_input;


  procedure Set_standard_input(
    opened_dev:  Device_Defs.opened_device)
    --
    --  Logic:
    --   1. Check that the new standard input is open,
    --      which implicitly checks that its an opened
    --      device.
```

```
    --   2. Check that that the new standard
    --      input has read rights.
    --   3. Set the new standard input.
  is
    stdin_untyped:  System.untyped_word;
      FOR stdin_untyped USE AT opened_dev'address;
  begin
    if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
      RAISE Device_Defs.device_not_open;

    elsif not Access_Mgt.Permits(
        AD     =>  stdin_untyped,
        rights =>  Device_Defs.read_rights) then
      RAISE System_Exceptions.insufficient_type_rights;

    else Process_Mgt.Set_process_globals_entry(
        slot   =>  Process_Mgt_Types.standard_input,
        value =>  stdin_untyped);
    end if;

  end Set_standard_input;


  function Get_standard_output
    return Device_Defs.opened_device
    --
    -- Logic:
    --    1. Get the process globals entry.
    --    2. Check that the new standard output is open,
    --       which implicitly checks that its an opened
    --       device.
    --    3. Check that the standard output has
    --       read rights.
    --    4. Return the new standard output.
  is
    stdout:          Device_Defs.opened_device;
    stdout_untyped:  System.untyped_word;
      FOR stdout_untyped USE AT stdout'address;
  begin
    stdout_untyped := Process_Mgt.
        Get_process_globals_entry(
            Process_Mgt_Types.standard_output);

    if not Byte_Stream_AM.Ops.Is_open(stdout) then
      RAISE Device_Defs.device_not_open;

    elsif not Access_Mgt.Permits(
        AD     =>  stdout_untyped,
        rights =>  Device_Defs.write_rights) then
      RAISE System_Exceptions.insufficient_type_rights;

    else
      RETURN stdout;

    end if;

  end Get_standard_output;


  procedure Set_standard_output(
    opened_dev:   Device_Defs.opened_device)
    --
    --  Logic:
    --   1. Check that the new standard output is
    --      open, which implicitly checks that its an
    --      opened device.
    --   2. Check that that the new standard output
    --      has write rights.
    --   3. Set the new standard output.
  is
    stdout_untyped:  System.untyped_word;
      FOR stdout_untyped USE AT
          opened_dev'address;
  begin
    if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
      RAISE Device_Defs.device_not_open;
```

```
    elsif not Access_Mgt.Permits(
        AD     =>  stdout_untyped,
        rights =>  Device_Defs.write_rights) then
      RAISE System_Exceptions.insufficient_type_rights;

    else Process_Mgt.Set_process_globals_entry(
        slot   =>  Process_Mgt_Types.standard_output,
        value  =>  stdout_untyped);
    end if;

  end Set_standard_output;


  function Get_standard_message
    return Device_Defs.opened_device
    --
    -- Logic:
    --     1. Get the process globals entry.
    --     2. Check that the standard message
    --        output is open, which implicitly
    --        checks that its an opened device.
    --     3. Check that the standard message
    --        output has write rights.
    --     4. Return the standard message output.
  is
    stdmsg:          Device_Defs.opened_device;
    stdmsg_untyped:  System.untyped_word;
      FOR stdmsg_untyped USE AT
          stdmsg'address;
  begin
    stdmsg_untyped := Process_Mgt.
        Get_process_globals_entry(
            Process_Mgt_Types.standard_message);

    if not Byte_Stream_AM.Ops.Is_open(stdmsg) then
      RAISE Device_Defs.device_not_open;

    elsif not Access_Mgt.Permits(
        AD     =>  stdmsg_untyped,
        rights =>  Device_Defs.write_rights)
        then
      RAISE System_Exceptions.insufficient_type_rights;

    else
      RETURN stdmsg;

    end if;
  end Get_standard_message;


  procedure Set_standard_message(
    opened_dev:  Device_Defs.opened_device)
    --
    --  Logic:
    --      1. Check that the new standard message
    --         output is open, which implicitly checks
    --         that its an opened device.
    --      2. Check that that the new standard
    --         message has write rights.
    --      3. Set the new standard message output.
  is
    stdmsg_untyped:  System.untyped_word;
      FOR stdmsg_untyped USE AT
          opened_dev'address;
  begin
    if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
      RAISE Device_Defs.device_not_open;

    elsif not Access_Mgt.Permits(
        AD     =>  stdmsg_untyped,
        rights =>  Device_Defs.write_rights) then
      RAISE System_Exceptions.insufficient_type_rights;

    else Process_Mgt.Set_process_globals_entry(
        slot   =>  Process_Mgt_Types.standard_message,
        value  =>  stdmsg_untyped);
    end if;
```

```
end Set_standard_message;


function Get_user_dialog
  return Device_Defs.opened_device
  --
  -- Logic:
  --    1. Get the process globals entry.
  --    2. Check that the user dialog is open,
  --       which implicitly checks that its an
  --       opened device.
  --    3. Check that the user dialog has
  --       read and write rights.
  --    4. Return the user dialog.
is
  user_dialog:         Device_Defs.opened_device;
  user_dialog_untyped: System.untyped_word;
    FOR user_dialog_untyped USE AT
        user_dialog'address;
begin
  user_dialog_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.user_dialog);

  if not Byte_Stream_AM.Ops.Is_open(user_dialog) then
    RAISE Device_Defs.device_not_open;

  elsif not Access_Mgt.Permits(
      AD =>     user_dialog_untyped,
      rights => Device_Defs.read_write_rights)
  then
    RAISE System_Exceptions.insufficient_type_rights;

  else
    RETURN user_dialog;

  end if;

end Get_user_dialog;


procedure Set_user_dialog(
  opened_dev:  Device_Defs.opened_device)
  --
  --  Logic:
  --   1. Check that the new user_dialog is open,
  --      which implicitly checks that its an opened
  --      device.
  --   2. Check that that the new user dialog has
  --      read and write rights.
  --   3. Set the new standard message.
is
  user_dialog_untyped:  System.untyped_word;
    FOR user_dialog_untyped USE AT
        opened_dev'address;
begin
  if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
    RAISE Device_Defs.device_not_open;

  elsif not Access_Mgt.Permits(
      AD     => user_dialog_untyped,
      rights => Device_Defs.read_write_rights)
  then
    RAISE System_Exceptions.insufficient_type_rights;

  else Process_Mgt.Set_process_globals_entry(
      slot  => Process_Mgt_Types.user_dialog,
      value => user_dialog_untyped);
  end if;

end Set_user_dialog;


function Get_home_directory
  return Directory_Mgt.directory_AD
  --
```

```
  -- Logic:
  --    1. Get the process globals entry for
  --       the "home directory."
  --    2. Check that the entry is a
  --       directory.
  --    3. Check that directory has read rights.
  --    4. Return the directory.
is
  dir:         Directory_Mgt.directory_AD;
  dir_untyped: System.untyped_word;
    FOR  dir_untyped USE AT
         dir'address;
begin
  dir_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.home_dir);

 if not Directory_Mgt.Is_directory(dir_untyped) then
   RAISE System_Exceptions.type_mismatch;

 else
   RETURN dir;

 end if;

end Get_home_directory;


function Get_current_directory
  return Directory_Mgt.directory_AD
  --
  -- Logic:
  --    1. Get the process globals entry.
  --    2. Check that the "current directory"
  --       is a directory.
  --    3. Return the current directory.
is
  dir:         Directory_Mgt.directory_AD;
  dir_untyped: System.untyped_word;
    FOR dir_untyped USE AT dir'address;
begin
  dir_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.current_dir);

  if not Directory_Mgt.Is_directory(dir_untyped) then
    RAISE System_Exceptions.type_mismatch;

  else
    RETURN dir;

  end if;

end Get_current_directory;


procedure Set_current_directory(
  dir:  Directory_Mgt.directory_AD)
  --
  --  Logic:
  --     1. Check that the "current directory" is
  --        a directory.
  --     2. Set the new current directory.
is
  dir_untyped:  System.untyped_word;
    FOR dir_untyped USE AT dir'address;
begin
  if not Directory_Mgt.Is_directory(dir_untyped) then
    RAISE System_Exceptions.type_mismatch;

  else Process_Mgt.Set_process_globals_entry(
      slot   => Process_Mgt_Types.current_dir,
      value  => dir_untyped);
  end if;

end Set_current_directory;
```

```
function Get_authority_list
  return Authority_List_Mgt.authority_list_AD
  --
  -- Logic:
  --    1. Get the process globals entry.
  --    2. Check that the entry is an authority list.
  --    3. Return the authority list.
is
  auth_list: Authority_List_Mgt.authority_list_AD;
  auth_list_untyped:  System.untyped_word;
    FOR auth_list_untyped USE AT auth_list'address;
begin
  auth_list_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.authority_list);

  if not Authority_List_Mgt.
      Is_authority_list(auth_list_untyped) then
          RAISE System_Exceptions.type_mismatch;

  else
    RETURN auth_list;

  end if;

end Get_authority_list;


procedure Set_authority_list(
  auth:  Authority_List_Mgt.authority_list_AD)
  --
  -- Logic:
  --   1. Check that "auth" is an authority list.
  --   2. Set the new authority list.
is
  auth_untyped:  System.untyped_word;
    FOR auth_untyped USE AT auth'address;
begin
  if not Authority_List_Mgt.Is_authority_list(
      auth_untyped) then
    RAISE System_Exceptions.Type_mismatch;

  else Process_Mgt.Set_process_globals_entry(
      slot  => Process_Mgt_Types.authority_list,
      value => auth_untyped);
  end if;

end Set_authority_list;


function Get_ID_list
  return Identification_Mgt.ID_list_AD
  --
  -- Logic:
  --    1. Get the process globals entry.
  --    2. Check that the entry is an ID list.
  --    3. Return the ID list entry.
is
  ID_list:         Identification_Mgt.ID_list_AD;
  ID_list_untyped: System.untyped_word;
    FOR ID_list_untyped USE AT ID_list'address;
begin
  ID_list_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.ID_list);

  if not Identification_Mgt.
      Is_ID_list(ID_list_untyped) then
    RAISE System_Exceptions.type_mismatch;

  else
    RETURN ID_list;

  end if;

end Get_ID_list;
```

```


function Get_command_name_space
  return Name_Space_Mgt.name_space_AD
  --
  -- Logic:
  --     1. Get the process globals entry.
  --     2. Check that the entry is a name space.
  --     3. Return the name space entry.
is
  cmd_name_space :        Name_Space_Mgt.
                                  name_space_AD;
  cmd_name_space_untyped: System.untyped_word;
    FOR cmd_name_space_untyped USE AT
        cmd_name_space'address;
begin
  cmd_name_space_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.cmd_name_space);

  if not Name_Space_Mgt.
      Is_name_space(cmd_name_space_untyped) then
    RAISE System_Exceptions.type_mismatch;

  else
    RETURN cmd_name_space;

  end if;

end Get_command_name_space;


procedure Set_command_name_space(
  ns:  Name_Space_Mgt.name_space_AD)
  --
  --  Logic:
  --     1. Check that "ns" is a name space.
  --     2. Set the new command name space.
is
  ns_untyped:  System.untyped_word;
    FOR ns_untyped USE AT
        ns'address;
begin
  if not Name_Space_Mgt.
     Is_name_space(ns_untyped) then
    RAISE System_Exceptions.type_mismatch;

  else Process_Mgt.Set_process_globals_entry(
      slot  => Process_Mgt_Types.cmd_name_space,
      value => ns_untyped);
  end if;

end Set_command_name_space;

function This_process
  return Process_Mgt_Types.process_AD
  --
  --  Logic:
  --     1. Get the process globals entry
  --        for the current process.
  --     2. Return the process.
is
  current_process: Process_Mgt_Types.process_AD;
  current_process_untyped:  System.untyped_word;
    FOR current_process_untyped USE AT
        current_process'address;
begin
  current_process_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.process);

  RETURN current_process;

end This_process;

function Get_parent_process
  return Process_Mgt_Types.process_AD
```

```
  --
  --  Logic:
  --      1. Get the process globals entry
  --         for the parent process.
  --      2. Return the parent process.
is
  parent_process:          Process_Mgt_Types.
                                   process_AD;
  parent_process_untyped:  System.untyped_word;
    FOR parent_process_untyped USE AT
        parent_process'address;
begin
  parent_process_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.creator);

  RETURN parent_process;

end Get_parent_process;

function This_job
  return Job_Types.job_AD
  --
  --  Logic:
  --      1. Get the process globals
  --         entry for the current job.
  --      2. Return the current job.
is
  current_job:          Job_Types.job_AD;
  current_job_untyped:  System.untyped_word;
    FOR current_job_untyped USE AT
        current_job'address;
begin
  current_job_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.job);

  RETURN current_job;

end This_job;


function This_session
  return Session_Types.session_AD
  --
  --  Logic:
  --      1. Get process globals entry
  --         for the current session.
  --      2. Return the current session.
is
  current_session:          Session_Types.session_AD;
  current_session_untyped:  System.untyped_word;
    FOR current_session_untyped USE AT
        current_session'address;
begin
  current_session_untyped := Process_Mgt.
      Get_process_globals_entry(
          Process_Mgt_Types.session);

  RETURN current_session;

end This_session;


function Get_process_name
  return System_Defs.text_AD
  --
  --  Logic:
  --      1. Return the name of the current process.
is
  name:          System_Defs.text_AD;
  name_untyped:  System.untyped_word;
    FOR name_untyped USE AT name'address;
begin
  name_untyped := Process_Mgt.
      Get_process_globals_entry(
           Process_Mgt_Types.name);
```

```
    RETURN name;

  end Get_process_name;



  procedure Set_process_name(
    name:  System_Defs.text)
    --
    --  Logic:
    --      1. Check that "name" is a valid text.
    --      2. Set the new process name.
  is
    name_untyped:  System.untyped_word;
      FOR name_untyped USE AT
          name'address;
  begin
    if name.length > name.max_length then
      RAISE System_Exceptions.bad_parameter;

    else
      Process_Mgt.Set_process_globals_entry(
          slot  => Process_Mgt_Types.name,
          value => name_untyped);
    end if;

  end Set_process_name;


end Process_Globals_Support_Ex;
```

X-A.6.10 `Symbol_Table_Ex` Package Specification

```
package Symbol_Table_Ex is
  --
  -- Function:
  --   Manages a symbol table for use by a compiler or
  --   other application.
  --
  --   Synchronizes concurrent access to the symbol
  --   table.
  --
  --   Symbol names can be no longer than
  --   "max_symbol_length" characters.
  --
  --   There is no limit on the number of symbols in
  --   the table; it is expanded as needed.
  --
  --   The symbol table is created empty at package
  --   initialization.
  --
  -- Notes:
  --   Nested blocks and symbols local to blocks are
  --   not supported.
  --
  -- Exceptions:
  --
  symbol_exists:   exception;
    -- "Add_symbol" was called with a symbol that is
    -- already in the table.
    --
  no_such_symbol:  exception;
    -- "Read_symbol_data" was called with a symbol
    -- that is not in the table.
    --
  name_too_long:   exception;
    -- "Add_symbol" or "Read_symbol_data" was called
    -- with a symbol name longer than
    -- "max_symbol_length".
    --
  max_symbol_length:  constant positive := 32;
    -- Maximum symbol length allowed.
  --
  -- History:
  --   11-24-87, Paul Schwabe: updated spec.
  --
  -- End of Header
  pragma external;

  type symbol_data is record
    -- This type defines the characteristics recorded
    -- for each symbol in the table.  No fields are
    -- defined for this example package.
    null;
  end record;


  procedure Add_symbol(
      name:  string;
        -- Name cannot be in use in the table. Name
        -- cannot be longer than "max_symbol_length".
      data:  symbol_data);
    --
    -- Function:
    --   Adds a symbol and its data to the symbol
    --   table.
    --
    -- Exceptions:
    --   symbol_exists
    --   name_too_long


  function Read_symbol_data(
      name:  string)
        -- Must name a symbol in the table. Name
        -- cannot be longer than "max_symbol_length".
    return symbol_data;
```

```
    --
    -- Function:
    --   Reads a symbol's data from the symbol table.
    --
    -- Exceptions:
    --   no_such_symbol
    --   name_too_long


end Symbol_Table_Ex;
```

X-A.6.11 `Symbol_Table_Ex` Package Body

```
with Object_Mgt,
     Semaphore_Mgt,
     System;

package body Symbol_Table_Ex is
  --
  -- Logic:
  --   The symbol table is implemented as an object
  --   containing an array.  Because the table is
  --   dynamically allocated, it can be expanded as
  --   needed.
  --
  --   The "symbol_table.lock" semaphore is used to
  --   exclude other processes while a process is
  --   accessing the table. All symbol table
  --   operations lock ("P") the semaphore before
  --   accessing the table, and unlock ("V") the
  --   semaphore before returning or propagating an
  --   exception.
  --
  -- Notes:
  --   A realistic implementation could be optimized
  --   for keyed retrieval using a hash table.  Such
  --   an implementation could use the same locking
  --   code.
  -- History:
  --   11-24-87, Paul Schwabe: updated code.
  --   11-25-87, Gary Taylor :  Added tagged comment lines.
  --
  -- End of Header

  use System;  -- Import arithmetic on type "ordinal".

  table_size: constant System.ordinal := 100;

  type symbol_name is array(
      1 .. max_symbol_length) of character;

  type symbol_entry is record
    name:  symbol_name;
    data:  symbol_data;
  end record;

  FOR symbol_entry USE
     record at mod 32;
  end record;

  type symbol_entry_array is array(
      System.ordinal range <>) of symbol_entry;

  type symbol_table_object(
      max_length:  System.ordinal) is record
    -- "max_length" is maximum number of entries in a
    -- full table.  Table can still grow by calling
    -- "Expand_symbol_table".
      length:  System.ordinal;
        -- Number of entries in use.
      lock:  Semaphore_Mgt.semaphore_AD;
        -- Used to lock symbol table while a process
        -- is accessing it.
      value:   symbol_entry_array(1 .. max_length);
        -- Entries 1 .. "length" contain symbol
        -- entries.
  end record;

  type symbol_table_AD is access symbol_table_object;
    pragma access_kind(symbol_table_AD, AD);

  symbol_table:  symbol_table_AD;
  procedure Expand_symbol_table is
    --
    -- Operation:
    --   Doubles the symbol table size.
    --
```

```
  --   "Expand_symbol_table" is normally called only
  --   when the symbol table is full.
  --
  --   Performs these steps:
  --   1. Resizes the symbol table object with space
  --      for twice as many entries.
  --   2. Changes the maximum length of the
  --      symbol table entry.
  --
  -- Notes:
  --   "Expand_symbol_table" is an internal
  --   procedure that must be called with the symbol
  --   table already locked via the associated
  --   semaphore!

  symbol_table_untyped: System.untyped_word;
    FOR symbol_table_untyped USE AT
        symbol_table'address;

  max_length_access:  System.ordinal;
    FOR max_length_access USE AT
        symbol_table.max_length'address;
begin
  Object_Mgt.Resize(
      obj  => symbol_table_untyped,
      size => 3 + (2 * symbol_table.max_length * (
          symbol_entry'size/32)));

  max_length_access := 2 * symbol_table.max_length;


end Expand_symbol_table;


procedure Add_symbol(
    name:  string;
    data:  symbol_data)
  --
  -- Logic:
  --   1. Surround everything else with a lock on
  --      "symbol_table.lock".  Release the lock
  --      on all return paths and exception paths.
  --   2. Check for "name" too long.
  --   3. Convert "name" to "fixed_width_name",
  --      padding with blanks.
  --   4. Search the table and raise an exception if
  --      the symbol is in the table.
  --   5. Otherwise, add the symbol to the end of
  --      the table, expanding the symbol table if
  --      it is full.
is
  fixed_width_name:  symbol_name := (others => ' ');
begin
  Semaphore_Mgt.P(symbol_table.lock);
    begin
      if name'length > max_symbol_length then
        RAISE name_too_long;

      else
        fixed_width_name(1 .. name'length) :=
            symbol_name(name);
        for i in 1 .. symbol_table.length loop
          if symbol_table.value(i).name =
              fixed_width_name then
            RAISE symbol_exists;
          end if;
        end loop;
        if symbol_table.length =
            symbol_table.max_length then
          Expand_symbol_table;
        end if;
        symbol_table.length := symbol_table.length + 1;
        symbol_table.value(symbol_table.length) :=
            symbol_entry'(fixed_width_name, data);
      end if;

    exception
```

```
      when others =>
        Semaphore_Mgt.V(symbol_table.lock);
        RAISE;
        -- Reraise exception that entered handler.
    end;

  Semaphore_Mgt.V(symbol_table.lock);


end Add_symbol;


function Read_symbol_data(
    name:  string)
  return symbol_data
  --
  -- Logic:

  --   1. Surround everything else with a lock on
  --      "symbol_table.lock".  Release the lock
  --      on all return paths and exception paths.

  --   2. Check for "name" too long.

  --   3. Convert "name" to "fixed_width_name",
  --      padding with blanks.

  --   4. Search the table.  If the symbol is found,
  --   return the symbol data.  Otherwise raise
  --   "no_such_symbol".
is
  fixed_width_name:  symbol_name := (others => ' ');
begin

  Semaphore_Mgt.P(symbol_table.lock);

  if name'length > max_symbol_length then
    RAISE name_too_long;

  else
    fixed_width_name(1 .. name'length) :=
        symbol_name(name);
    for i in 1 .. symbol_table.length loop
      if symbol_table.value(i).name =
          fixed_width_name then
        Semaphore_Mgt.V(symbol_table.lock);
        RETURN symbol_table.value(i).data;

      end if;
    end loop;
      RAISE no_such_symbol;

  end if;

  -- This call to "V" is never reached in the
  -- current implementation.  The call is included
  -- as a safeguard in case code changes make it
  -- reachable.
  Semaphore_Mgt.V(symbol_table.lock);

  exception
    when others =>
      Semaphore_Mgt.V(symbol_table.lock);
      RAISE;  -- Reraise exception
              -- that entered handler.

end Read_symbol_data;


-- PACKAGE INITIALIZATION
begin
  symbol_table := new symbol_table_object(
      table_size);
  symbol_table.length := 0;
    -- Symbol table initially has space for 100
    -- entries with 0 in use.
```

```
    symbol_table.lock := Semaphore_Mgt.
        Create_semaphore;
      -- Lock initially indicates table is available.
      -- First "P" on lock will succeed.

end Symbol_Table_Ex;

```

X-A.6.12 Word_Processor_Ex Package Specification

```
package Word_Processor_Ex is
  --
  -- Function:
  --   This example shows how a word processor with a
  --   spelling checker can use processes and events.
  --
  -- End of Header
  pragma external;


  procedure Word_processor;
    --
    -- Function:
    --   Does word processing.
    --
    --   Gets its arguments from the command line.
    --
    --   Includes a concurrent spelling checker.


end Word_Processor_Ex;
```

X-A.6.13 Word_Processor_Ex Package Body

```
with Conversion_Support_Ex,
     Event_Mgt,
     Process_Globals_Support_Ex,
     Process_Mgt,
     Process_Mgt_Types,
     System;

package body Word_Processor_Ex is
  --
  -- Logic:
  --   This example shows how a word processor with a
  --   concurrent spelling checker uses processes
  --   and events.
  --
  --   The "Word_processor" procedure spawns a
  --   separate process to execute the
  --   "Spelling_checker" procedure. Communication
  --   between the two processes is entirely via
  --   events.
  --
  --   When a word is entered by the word processor
  --   user, the word processor signals a 'word' event
  --   to the spelling checker process.  That event
  --   has these
  --   fields:
  --
  --     "event"          - "word_event_value".
  --
  --     "message.offset" - Location of word to check,
  --                        encoded as a 32-bit
  --                        "word_record".
  --
  --     "message.AD"     - AD to word processor
  --                        process that is signalling
  --                        the event.
  --
  --     "destination"    - AD to spelling checker
  --                        process that receives
  --                        the event.
  --
  --   Inclusion of an AD to the process that signals
  --   the event allows a future implementation to use
  --   the spelling checker process as a server for
  --   multiple client processes.
  --
  --   If a word is misspelled, the spelling checker
  --   signals a 'spelling error' event to the process
  --   that requested the spelling check.
  --   That event has these fields:
  --
  --    "event"          - spelling_error_event_value.
  --
  --    "message.offset" - Location of word that was
  --                       checked, encoded as a 32-bit
  --                       "word_record".
  --
  --    "message.AD"     - Not used.  In this
  --                       implementation,
  --                       is "System.null_word".
  --
  --   "destination"    - AD to the word processor
  --   process that signalled the word to the spelling
  --   checker.
  --
  --   The word processor handles spelling error
  --   events with the "Spelling_error_handler"
  --   procedure.
  --
  -- Notes:
  --   The "word_record" scheme of communicating words
  --   to be checked is probably inadequate for an
  --   implementation of the spelling checker as a
  --   general server that can be used by multiple
  --   concurrent applications.
```

```
--
-- History:
--   11-24-87, Paul Schwabe: updated code.
--   11-25-87, Gary Taylor :  Added tagged comment lines.
--
-- End of Header

use System;  -- Import operations on ordinal types.

type word_record is record
  -- This type encodes a word location into 32 bits,
  -- allowing a word location to be transmitted
  -- using the "message.offset" field when an event
  -- is signalled.  The word processor and spelling
  -- checker are presumed to share a two-dimensional
  -- array containing the text being edited.  Words
  -- are presumed to not break across lines of the
  -- array.  A word location can thus be specified
  -- as a line number, a starting column number, and
  -- an ending column number.  The encoding limits
  -- line numbers to the range 0 .. 65_535 and
  -- column numbers to the range 0 .. 255.
  line:       System.short_ordinal;
  start_col:  System.byte_ordinal;
  end_col:    System.byte_ordinal;
end record;

FOR word_record USE
  record at mod 32;
    line      at 0 range  0 .. 15;
    start_col at 0 range 16 .. 23;
    end_col   at 0 range 24 .. 31;
  end record;


-- << Event Values Used >>
--
-- The following local events can use the same event
-- value without conflict because they are always
-- signalled to different processes.

word_event_value:
  constant Event_Mgt.event_value := Event_Mgt.user_1;
  -- Local event signalled to spelling checker for
  -- each word to be checked.

spelling_error_event_value:
    constant Event_Mgt.event_value :=
        Event_Mgt.user_1;
  -- Local event signalled to client process for
  -- each misspelled word.


procedure Spelling_checker(
    param_buffer:  System.address;
      -- Not used but required for process's initial
      -- procedure.
    param_length:  System.ordinal)
      -- Not used but required for process's initial
      -- procedure.
  -- Operation:
  --    Loops doing these steps:
  --      1. Wait for a word event.
  --      2. Check the word's spelling.
  --      3. If the word is misspelled, signal a
  --         spelling error event to whatever
  --         process requested the check.
is
  word_event:      Event_Mgt.action_record;
    -- Receives each word to be checked.
  current_word:    word_record;
  FOR current_word USE AT word_event.
      message.offset'address;
    -- Overlay used to extract word location.,
  word_mispelled:  boolean;
begin
  loop
```

```
    Event_Mgt.Wait_for_any(
        events => (word_event_value => true,
           others => false),
        action => word_event);

    -- Code`to check spelling of current word goes
    -- here. The "word_mispelled" flag is a stand-in
    -- for whatever conditional expression indicates
    -- a mispelled word.

    if word_mispelled then
      Event_Mgt.Signal(Event_Mgt.action_record'(
          event       => spelling_error_event_value,
          message     => (
              offset => word_event.message.offset,
              AD      => System.null_word),
          destination => word_event.message.AD));
    end if;

  end loop;

end Spelling_checker;
pragma subprogram_value(Process_Mgt.Initial_proc,
    Spelling_checker);


procedure Spelling_error_handler(
    action:  Event_Mgt.action_record)
  --
  -- Operation:
  --   Handler invoked for each 'spelling error'
  --   event.
is
  misspelled_word:  word_record;
  FOR misspelled_word
      USE AT action.message.offset'address;
    -- Overlay used to extract word location.
begin
  -- Code to handle misspelled word goes here.  For
  -- example, this code could highlight the
  -- misspelled word on the display and ring the
  -- terminal's bell.

  null;
end Spelling_error_handler;
  pragma subprogram_value(
      Event_Mgt.Event_handler,
      Spelling_error_handler);


procedure Word_processor
  --
  -- Logic:
  --   1. Retrieve an AD for this process, to be
  --      passed to the spelling checker so it will
  --      know what process to signal if a word is
  --      misspelled.
  --
  --   2. Create the spelling checker process.
  --
  --   3. Establish a handler for the spelling error
  --      local event and enable the event.  Save the
  --      previous event status.
  --
  --   4. Loop, doing word processing.  For each
  --      word that is entered, signal the word event
  --      to the spelling checker process.
  --
  --   5. When word processing is done, terminate and
  --      deallocate the spelling checker process and
  --      restore the previous event status for the
  --      spelling error local event.
is
  spelling_checker_process:
      Process_Mgt_Types.process_AD;
    -- Process executing "Spelling_checker".
```

```
  child_termination_event_value:
      constant Event_Mgt.event_value :=
          Event_Mgt.user_2;
    -- Local event signalled when spelling checker
    -- process terminates.

  child_termination_event:  Event_Mgt.action_record;
    -- Action record used to receive spelling checker
    -- process's termination event.

  this_process_untyped:  System.untyped_word;
    -- Process executing "Word_processor",
    -- as an "untyped_word".

  word_event:  Event_Mgt.action_record;
    -- Used to signal each word to be checked.
  current_word:  word_record;
  FOR current_word
      USE AT word_event.message.offset'address;
    -- Overlay used for word location.

  old_event_status:  Event_Mgt.event_status;
    -- Saves previous event status for the
    -- spelling_error local event, so the previous
    -- status can be restored before exit.

begin
  this_process_untyped :=
      Process_Mgt.Get_process_globals_entry(
          Process_Mgt_Types.process);

  spelling_checker_process := Process_Mgt.
     Spawn_process(
      init_proc   =>
          Spelling_checker'subprogram_value,
      term_action => (
          event       =>
              child_termination_event_value,
          message     => System.null_address,
            -- Not used.
          destination => this_process_untyped));

  old_event_status := Event_Mgt.
      Establish_event_handler(
          event  => spelling_error_event_value,
          status => (
              handler =>
                 Spelling_error_handler'
                     subprogram_value,
                 state   => Event_Mgt.enabled,
                 interrupt_system_call => false));

  loop
    -- Presume that control exits the loop when a
    -- user quits the word processor.

    -- Code to do word processing goes here.  For
    -- each word entered by the user,
    -- the following code is executed:

    word_event.event := word_event_value;
    word_event.message.AD := this_process_untyped;

    -- Code goes here to assign "current_word" which
    -- overlays "word_event.message.offset".

    word_event.destination :=
        Conversion_Support_Ex.Untyped_from_process(
            spelling_checker_process);
    Event_Mgt.Signal(word_event);

  end loop;

  << QUIT >>  -- Presume control reaches this point
              -- when a user exits the word
              -- processor.
```

```
    Event_Mgt.Signal(Event_Mgt.action_record'(
        event       => Event_Mgt.termination,
        message     => System.null_address,
          -- No message.
        destination => Conversion_Support_Ex.
                               Untyped_from_process(
                           spelling_checker_process)));
    Event_Mgt.Wait_for_any(
        events => (
            child_termination_event_value => true,
            others => false),
      action => child_termination_event);
    Process_Mgt.Deallocate(spelling_checker_process);

    old_event_status := Event_Mgt.
        Establish_event_handler(
            event  => spelling_error_event_value,
            status => old_event_status);
      -- Reestablish previous event status.
      -- Value returned is never used.

  end Word_processor;


end Word_Processor_Ex;
```

X-A.6.14 View_device_main Procedure

```
with CL_Defs,
     Command_Handler,
     Device_Defs,
     Environment_Mgt,
     System,
     System_Defs,
     VD_Commands,
     VD_Devices,
     VD_Defs;

procedure View_device_main
  --
  -- Function:
  --   Main program for "view.device" utility
  --   (Command-Oriented Program Example).
  --
  --   The procedure "View_device_main" is
  --   called from CLEX.  "View_device_main"
  --   performs the top-level processing for the
  --   "view.device" example utility.
  --
  -- History:
  --   10-08-87, William A. Rohm:  Written.
  --   11-17-87, WAR:              Revised.
  --
is

  -- Variables:
  --
  command:  System.short_ordinal;
    -- Index of current command (in current
    -- command set).

  command_name:  System_Defs.text(CL_Defs.max_name_sz);
    -- Name of current command (in current
    -- command set).

  current_cmd_odo:  Device_Defs.opened_device :=
      Command_Handler.Open_invocation_command_processing;
    -- Current opened command input device,
    -- initially the invocation command.

  device_name:  System_Defs.text(256);
    -- Pathname of viewed device.

  device_opened:  boolean;
    -- Returned true from
    -- "VD_Devices.Open_device" if device
    -- successfully opened.

  processing_runtime:  boolean := false;
    -- True if currently processing runtime
    -- commands, false if processing startup
    -- commands.

use System;       -- to import = for
                  -- System.short_ordinal

begin

  VD_Devices.Open_program_window;

  -- Get ":device" pathname:
  --
  Command_Handler.Get_string(
      cmd_odo    => current_cmd_odo,
      arg_number => 1,
      arg_value  => device_name);

  -- Close invocation command processing:
  --
  Command_Handler.Close(current_cmd_odo);


```

```
-- Open startup command input:
--
current_cmd_odo :=
    Command_Handler.Open_startup_command_processing(
        cmd_set => VD_Defs.main_cmd_set);


-- Main processing loop:
--
loop

  Command_Handler.Get_command(
      cmd_odo  => current_cmd_odo,
      prompt   => VD_Defs.main_prompt,
      cmd_id   => command,
      cmd_name => command_name);

  case command is
    when VD_Defs.main_change_ID =>
      Command_Handler.Get_string(
          cmd_odo    => current_cmd_odo,
          arg_number => 1,
          arg_value  => device_name);

      VD_Devices.device_info_valid := false;

    when VD_Defs.main_list_ID =>

      declare
        ops:  boolean;
          -- Returned ":operations" parameter.

      begin
        -- Get ":operations" parameter:
        --
        ops := Command_Handler.Get_boolean(
            cmd_odo    => current_cmd_odo,
            arg_number => 1);


        -- Display device information:
        --
        VD_Commands.Display_device_info(
            device_name => device_name,
            operations  => ops);

      end;


    when VD_Defs.main_access_ID =>

      declare
        open_mode:  System.short_ordinal;
          -- Enumeration index value of "access.device" method.

      begin
        -- Get desired open mode:
        --
        open_mode :=
            Command_Handler.Get_enumeration_index(
                cmd_odo    => current_cmd_odo,
                arg_number => 1);

        -- Open device:
        --
        device_opened := VD_Devices.Open_device(
            device_name => device_name,
            open_mode   => Device_Defs.
                open_mode'val(open_mode));
      end;

      if device_opened then
        -- Change to "access" command set:
        --
        Command_Handler.Change_cmd_set(
            cmd_odo      => current_cmd_odo,
            cmd_set_name => VD_Defs.access_cmd_set);
```

```

          VD_Commands.Process_access_commands(
              cmd_odo => current_cmd_odo);

          -- Return to "main" command set:
          --
          Command_Handler.Change_cmd_set(
              cmd_odo      => current_cmd_odo,
              cmd_set_name => VD_Defs.main_cmd_set);

        end if;     -- if device_opened

      when VD_Defs.main_exit_ID =>

        if processing_runtime then
          EXIT;
        else

          -- Close invocation command input
          -- device:
          --
          Command_Handler.Close(current_cmd_odo);

          -- Open runtime command processing:
          --
          current_cmd_odo :=
              Command_Handler.Open_runtime_command_processing(
                  cmd_set => VD_Defs.main_cmd_set);

          processing_runtime := true;

        end if;

      when others =>
        null;

    end case;

  end loop;

  if device_opened then
    VD_Devices.Close_device;
  end if;

  -- Close runtime command input device:
  --
  Command_Handler.Close(current_cmd_odo);

  -- Close program window:
  --
  VD_Devices.Close_program_window;

end View_device_main;
```

X-A.6.15 VD_Defs Package Specification

```
with System,
     System_Defs,
     Terminal_Defs;

package VD_Defs is
  --
  -- Function:
  --   Contains definitions for the constants in
  --   the Example Utility.
  --
  -- History:
  --   10-08-87, William A. Rohm:  Written.
  --   11-16-87, WAR:              Revised.
  --
  -- End of Header


  -- Constants:
  --
  program_window_size:
      Terminal_Defs.point_info := (80,20);
    -- Size of program's window, in columns
    -- and rows.

  program_buffer_size:
      Terminal_Defs.point_info := (80,20);
    -- Size of program window's buffer.

  program_window_pos:
      Terminal_Defs.point_info := (1,1);
    -- Position of program's window on
    -- terminal (upper left corner).

  main_cmd_set_str:  constant string := "$OEO/main";
    -- String value of main command set's
    -- pathname.

  main_cmd_set:  System_Defs.text(
      main_cmd_set_str'length) := (
          main_cmd_set_str'length,
          main_cmd_set_str'length,
          main_cmd_set_str);
    -- Pathname of main command set.


  access_cmd_set_str:  constant string := "$OEO/access";
    -- String value of "device access" command
    -- set's pathname.

  access_cmd_set:  System_Defs.text(
      access_cmd_set_str'length) := (
          access_cmd_set_str'length,
          access_cmd_set_str'length,
          access_cmd_set_str);
    -- Pathname of "device access" command set.


  main_prompt_str:  constant string := "view.device> ";
    -- String value of prompt for "main" command
    -- set.

  main_prompt:  System_Defs.text(
      main_prompt_str'length) := (
          main_prompt_str'length,
          main_prompt_str'length,
          main_prompt_str);
    -- "main" prompt's text.


  access_prompt_str:  constant string :=
      "access.device> ";
    -- String value of prompt for "access"
    -- command set.

```

```
  access_prompt:  System_Defs.text(
      access_prompt_str'length) := (
          access_prompt_str'length,
          access_prompt_str'length,
          access_prompt_str);
    -- "access" prompt's text.


  -- Command and Argument Indexes:
  --
    main_change_ID:  constant System.short_ordinal := 1;
    main_list_ID:    constant System.short_ordinal := 2;
    main_access_ID:  constant System.short_ordinal := 3;
    main_exit_ID:    constant System.short_ordinal := 4;
      -- *Main* command set command index values.

    input_index:          constant
        System.short_ordinal := 1;
    output_index:         constant
        System.short_ordinal := 2;
    input_partial_index:  constant
        System.short_ordinal := 3;
    input_output_index:   constant
        System.short_ordinal := 4;
      -- For "access.device  :open_mode"; the
      -- argument's enumeration index values.


    access_read_ID:   constant System.short_ordinal := 1;
    access_write_ID:  constant System.short_ordinal := 2;
    access_exit_ID:   constant System.short_ordinal := 3;
      -- *access* command set's
      -- command index values.

    read_length_arg:    constant
        System.short_ordinal := 1;
    read_position_arg:  constant
        System.short_ordinal := 2;
    read_offset_arg:    constant
        System.short_ordinal := 3;
      -- Argument index values for "read".

    write_position_arg:  constant
        System.short_ordinal := 1;
    write_offset_arg:    constant
        System.short_ordinal := 2;
      -- Argument index values for "write".

end VD_Defs;
```

X-A.6.16 VD_Commands Package Specification

```
with Device_Defs,
     System_Defs;

package VD_Commands is
  --
  -- Function:
  --   Contains operations related to processing
  --   "view.device" "access" command set's
  --   commands.
  --
  -- History:
  --   10-08-87, William A. Rohm:  Written.
  --   11-17-87, WAR:              Revised.
  --
  -- End of Header


  procedure Display_device_info(
    device_name:  System_Defs.text;
      -- Pathname of device.
    operations:   boolean);
      -- If true, displays "Byte_Stream_AM.Ops"
      -- operations supported by "device_name".
    --
    -- Function:
    --   Calls "VD_Devices.Get_device_info",
    --   then displays the returned device
    --   information record.


  procedure Process_access_commands(
      cmd_odo:  Device_Defs.opened_device);
        -- Opened command input device.
    --
    -- Function:
    --   Processes the "access" command set.


end VD_Commands;
```

X-A.6.17 VD_Commands Package Body

```
with Byte_Stream_AM,
     CL_Defs,
     Command_Handler,
     Device_Defs,
     System,
     System_Defs,
     Text_Mgt,
     VD_Defs,
     VD_Devices;

package body VD_Commands is
  --
  -- Function:
  --   Contains operations related to processing
  --   "view.device" "access" command set.
  --
  -- History:
  --   10-08-87, William A. Rohm:  Written.
  --   11-17-87, WAR:              Revised.
  --
  -- End of Header


  procedure Display_device_info(
    device_name:  System_Defs.text;
    operations:   boolean)
    --
    -- Logic:
    --   1. Check for valid device info record; get it
    --      if not valid
    --   2. Display common device info values
    --   3. Display BSAM device info values
    --   4. If "operations" is true, display supported
    --      ops

  is

    procedure Write_info(
        info_string:  string)
          -- String value to be written.
    is
    --
    -- Function:
    --   Display string value, followed by a linefeed.

      info_text:  System_Defs.text(32);
        -- Text value of various values' "'image"s.
    begin

      -- Make a text value of "info_string":
      --
      Text_Mgt.Set(
          dest   => info_text,
          source => info_string);

      -- Add a linefeed:
      --
      Text_Mgt.Append(
          dest   => info_text,
          source => Standard.ASCII.LF);

      -- Write text to the program's window:
      --
      Byte_Stream_AM.Ops.Write(
          opened_dev => VD_Devices.program_window,
          buffer_VA  => info_text'address,
          length     => System.ordinal(info_text.length));

    end Write_info;


  begin

   -- Check for valid "device_info":
```

```
    --
    if not VD_Devices.device_info_valid then
      VD_Devices.Get_device_info(
          device_name => device_name);
    end if;


     -- Display node id:
     --
     Write_info(
         info_string => "   Node ID:");

     Write_info(
         info_string => System_Defs.node_ID'image(
             VD_Devices.device_info.common_info.node));


     -- Display access methods supported:
     --
     Write_info(
         info_string => "   Access Methods Supported:");

     for i in Device_Defs.access_method'first ..
              Device_Defs.access_method'last loop

       if VD_Devices.device_info.
           common_info.acc_methods_supp(i) then

         Write_info(
             info_string => Device_Defs.
                 access_method'image(i));
       end if;

     end loop;


     -- Display open modes supported:
     --
     Write_info(
         info_string => "   Supported Open Modes:");

     for i in Device_Defs.open_mode'first ..
              Device_Defs.open_mode'last loop

       if VD_Devices.device_info.
           common_info.open_modes_supp(i) then

         Write_info(
             info_string => Device_Defs.
                 open_mode'image(i));
       end if;

     end loop;


     -- Display "store supported" boolean:
     --
     Write_info(
         info_string => "   Data written to device can be read back:");

     Write_info(
         info_string => boolean'image(
             VD_Devices.device_info.
                 common_info.store_supp));


     -- Display "is interactive" boolean:
     --
     Write_info(
         info_string => "   Device is interactive is:");

     Write_info(
         info_string => boolean'image(
             VD_Devices.device_info.
                 common_info.is_interactive));
```

```
    -- Display byte-stream operations supported;
    --
    if operations then
      Write_info(
          info_string => "   Supported Byte Stream Operations:");

      for i in Byte_Stream_AM.bsam_operation'first ..
               Byte_Stream_AM.bsam_operation'last loop

        if VD_Devices.device_info.bsam_ops_supp(i) then

          Write_info(
              info_string => Byte_Stream_AM.
                  bsam_operation'image(i));
        end if;

      end loop;

    end if;

  end Display_device_info;


  procedure Process_access_commands(
      cmd_odo:  Device_Defs.opened_device)

  is
    command:  System.short_ordinal;
      -- Index of current command (in current
      -- command set).

    command_name:  System_Defs.text(CL_Defs.max_name_sz);
      -- Name of current command (in current
      -- command set).

    length:  CL_Defs.CL_range;
      -- Length of displayed bytes for "read :length".

    position:  System.short_ordinal;
      -- Index of "read/write :position" argument's value.

    offset:  integer;
      -- Value of "read/write :offset" argument.

  begin

    -- Command processing loop:
    --
    loop

      Command_Handler.Get_command(
          cmd_odo  => cmd_odo,
          prompt   => VD_Defs.access_prompt,
          cmd_id   => command,
          cmd_name => command_name);


      case command is

        when VD_Defs.access_read_ID =>

          -- Get ":length" argument:
          --
          length := Command_Handler.Get_range(
              cmd_odo    => cmd_odo,
              arg_number => VD_Defs.read_length_arg);

          -- Get ":position" argument:
          --
          position := Command_Handler.
              Get_enumeration_index(
                  cmd_odo    => cmd_odo,
                  arg_number => VD_Defs.read_position_arg);

          -- Get ":offset" argument:
          --
          offset := Command_Handler.Get_integer(
```

```
              cmd_odo    => cmd_odo,
              arg_number => VD_Defs.read_offset_arg);

          -- Read and display bytes:
          --
          -- TBD

        when VD_Defs.access_write_ID =>

          -- Get ":position" argument:
          --
          position := Command_Handler.
              Get_enumeration_index(
                  cmd_odo    => cmd_odo,
                  arg_number => VD_Defs.write_position_arg);

          -- Get ":offset" argument:
          --
          offset := Command_Handler.Get_integer(
              cmd_odo    => cmd_odo,
              arg_number => VD_Defs.write_offset_arg);

          -- Get bytes and write to device:
          --
          -- TBD

        when VD_Defs.access_exit_ID =>
            EXIT;

        when others =>
          null;

      end case;

    end loop;

  end Process_access_commands;

end VD_Commands;
```

X-A.6.18 VD_Devices Package Specification

```
with Byte_Stream_AM,
     Device_Defs,
     Long_Integer_Defs,
     System,
     System_Defs;

package VD_Devices is
  --
  -- Function:
  --   Contains all operations related to the
  --   viewed device and the windows.
  --
  --   This package contains calls to open and
  --   close the program's windows, and calls to
  --   read and write bytes to and from the
  --   viewed device.
  --
  -- History:
  --   10-08-87, William A. Rohm:  Written.
  --   11-17-87, WAR:              Revised.
  --
  -- End of Header


  -- Variables:
  --
  program_window:  Device_Defs.opened_device;
    -- Utility's window, for accepting commands
    -- and displaying data.


  opened_device:  Device_Defs.opened_device :=
      System.null_word;
    -- Opened viewed device.

  device_info:  Byte_Stream_AM.device_info;
    -- Device information record for
    -- "Byte_Stream_AM".

  device_info_valid:  boolean := false;
    -- Whether the device information record is valid.


  procedure Open_program_window;
    --
    -- Function:
    --   Open the program's window on the
    --   current terminal.


  procedure Close_program_window;
    --
    -- Function:
    --   Closes the program's main window, and
    --   any opened "::window" windows.


  procedure Get_device_info(
    device_name:  System_Defs.text);
    --
    -- Function:
    --   Calls "Byte_Stream_AM.Get_device_info" to set
    --   "VD_Devices.device_info" information record.


  function Open_device(
      device_name:  System_Defs.text;
        -- Pathname of device to be opened.
      open_mode:    Device_Defs.open_mode)
        -- Open mode for device.
    return boolean;
        -- True if device successfully opened.
    --
    -- Function:
```

```
    --   Opens given device with
    --   "Byte_Stream_AM.Open_by_name",
    --   returning true if successful.
    --
    --   Sets this package's "opened_device"
    --   variable; "System.null_word" if
    --   inaccessible.


  procedure Read_bytes(
      length:       System.ordinal;
        -- Number of bytes to be read and
        -- displayed.
      position:     Byte_Stream_AM.position_mode;
        -- Position from which "offset" is measured.
      offset:       integer;
        -- Offset of first byte to be read and
        -- displayed.
      bytes:    out System_Defs.text);
        -- Bytes read from device.
    --
    -- Function:
    --   Reads and displays bytes from the opened
    --   device.


  procedure Write_bytes(
      position:  Byte_Stream_AM.position_mode;
        -- Position from which "offset" is measured.
      offset:    System.ordinal;
        -- Offset of first byte to be written to
        -- device.
      bytes:     System_Defs.text);
        -- Bytes to be written to device.
    --
    -- Function:
    --   Reads and displays bytes from the opened
    --   device.


  procedure Close_device;
    --
    -- Function:
    --   Closes opened device with
    --   "Byte_Stream_AM.Close".

end VD_Devices;
```

X-A.6.19 `VD_Devices` Package Body

```
with Access_Mgt,
     Byte_Stream_AM,
     Device_Defs,
     Directory_Mgt,
--     Example_Messages,
     Object_Mgt,
     Process_Mgt,
     Process_Mgt_Types,
     System,
     System_Defs,
     System_Exceptions,
     Terminal_Defs,
     Unchecked_Conversion,
     VD_Defs,
     Window_Services;


package body VD_Devices is
  --
  -- History:
  --   10-08-87, William A. Rohm:  Written.
  --   11-17-87, WAR:              Revised.
  --
  -- End of Header



  procedure Open_program_window
    --
    -- Logic:
    --  1.  Gets device AD to underlying terminal.
    --  2.  Opens and assigns "program_window".
  is
    old_opened_window:     Device_Defs.opened_device;
    old_window:            Device_Defs.device;
    underlying_terminal:   Device_Defs.device;

  begin

    -- Assume standard input, on entry, is from
    -- an opened window:
    --
    old_opened_window :=
        Process_Mgt.Get_process_globals_entry(
            Process_Mgt_Types.standard_input);


    -- Get device object of standard input
    -- window:
    --
    old_window :=
        Byte_Stream_AM.Ops.Get_device_object(
            old_opened_window);


    -- Get device AD of standard input window's
    -- terminal:
    --
    underlying_terminal :=
        Window_Services.Ops.Get_terminal(
            old_window);


    -- Create program window:
    --
    program_window := Window_Services.Ops.Create_window(
        terminal            => underlying_terminal,
        pixel_units         => false,  -- characters, not pixels
        fb_size             => VD_Defs.program_buffer_size,
        desired_window_size => VD_Defs.program_window_size,
        window_pos          => VD_Defs.program_window_pos,
        view_pos            => Terminal_Defs.point_info'(1,1));

  end Open_program_window;
```

```


procedure Close_program_window
  --
  -- Logic:
  --   1.  Close the program window.
is
begin

  Window_Services.Ops.Destroy_window(program_window);

end Close_program_window;


procedure Get_device_info(
    device_name:  System_Defs.text)

is
  device:  Device_Defs.device;
    -- Device.

  device_untyped:  System.untyped_word;
    FOR device_untyped USE AT device'address;
    -- Device as an untyped word.

begin

  begin

    device_untyped := Directory_Mgt.Retrieve(
        name => device_name);

    device_info :=
        Byte_Stream_AM.Ops.Get_device_info(
            dev => device);

    device_info_valid := true;

  exception
    when Directory_Mgt.no_access =>
      RAISE;  -- msg no_access

    when others => RAISE;

  end;

end Get_device_info;


function Open_device(
    device_name:  System_Defs.text;
    open_mode:    Device_Defs.open_mode)
  return boolean
  --
  -- Logic:
  --   1. Check for allowed open mode
  --   2. Attempt "BSAM_AM.Open_by_name"
  --   3. If successful, assign
  --      "opened_device", return true;
  --      otherwise, assign "opened_device"
  --      null, return false
is
  successful:  boolean := false;
    -- Returned true if successfully opens
    -- device.
begin

  if not device_info_valid then
    Get_device_info(device_name);
  end if;

  if device_info_valid and
      device_info.common_info.
          open_modes_supp(open_mode) then

    -- Try to open device:
    --
```

```
      begin
        opened_device := Byte_Stream_AM.Open_by_name(
            name         => device_name,
            input_output => open_mode,
            allow        => Device_Defs.anything,
            block        => true);

      successful := true;

      exception
        when others =>
            opened_device := System.null_word;
      end;

    end if;     -- if valid and open_mode
                -- supported

    return successful;

  end Open_device;



  procedure Read_bytes(
      length:       System.ordinal;
      position:     Byte_Stream_AM.position_mode;
      offset:       integer;
      bytes:    out System_Defs.text)
  is
    byte_position:  Long_Integer_Defs.long_integer;
      -- Byte pointer position, returned from
      -- "Byte_Stream_AM.Ops.Set_position".

    bytes_read:  System.ordinal;
      -- Number of bytes actually read.

  use System;    -- to import "/=" for System.ordinal

  begin

      byte_position := Byte_Stream_AM.Ops.Set_position(
          opened_dev => opened_device,
          pos        => Long_Integer_Defs.
              Convert_to_long_integer(
                  number => offset),
          mode       => position);

      bytes_read := Byte_Stream_AM.Ops.Read(
          opened_dev => opened_device,
          buffer_VA  => bytes'address,
          length     => System.ordinal(offset),
          block      => false);

      if integer(bytes_read) = offset then

        bytes_read := Byte_Stream_AM.Ops.Read(
            opened_dev => opened_device,
            buffer_VA  => bytes'address,
            length     => bytes'size/8,
            block      => false);

        if bytes_read /= length then
          bytes.length := System_Defs.text_length(bytes_read);
        end if;

      end if;

  end Read_bytes;


  procedure Write_bytes(
      position:  Byte_Stream_AM.position_mode;
      offset:    System.ordinal;
      bytes:     System_Defs.text)
  is
    bytes_read:  System.ordinal;
      -- Number of bytes actually read.
```

```

   use System;    -- import "=" for System.ordinal;

   begin

       bytes_read := Byte_Stream_AM.Ops.Read(
           opened_dev => opened_device,
           buffer_VA  => bytes'address,
           length     => offset,
           block      => false);

       if bytes_read = offset then

         bytes_read := Byte_Stream_AM.Ops.Read(
             opened_dev => opened_device,
             buffer_VA  => bytes'address,
             length     => bytes'size/8,
             block      => false);

       end if;

   end Write_bytes;


   procedure Close_device
   is
   begin
     Byte_Stream_AM.Ops.Close(opened_device);
   end Close_device;

 end VD_Devices;
```

X-A.7 Type Manager Services

X-A.7.1 Acct_main_ex Procedure

Main procedure of the account manager test driver.

```
-------------------------------------------------------------
-------------------------------------------------------------
--                                                         --
--                  COMMAND DEFINITIONS                    --
--                                                         --
-------------------------------------------------------------
-------------------------------------------------------------


--*D* manage.commands
--*D*
--*D*
--*D*
--*D*   create.invocation_command
--*D*   end
--*D*
--*D*
--*D*
--*D*   create.runtime_command_set :cmd_def = acct_cmds \
--*D*                              :prompt  = "ACCT_MGT> "
--*D*
--*D*     define.command :cmd_name = create
--*D*       set.description :text = "
--*D*           -- Create a new account with an initial balance.
--*D*           "
--*D*
--*D*       define.argument :arg_name = init_balance \
--*D*                       :type     = integer
--*D*         set.description :text = "
--*D*             -- Initial balance of an account.
--*D*             -- Must be between 0 an 100000.
--*D*             "
--*D*         set.bounds         :value = 0..100000
--*D*         set.mandatory
--*D*       end
--*D*     end
--*D*
--*D*
--*D*     define.command :cmd_name = cstore
--*D*       set.description :text = "
--*D*           -- Create and store a new account in one step.
--*D*           "
--*D*
--*D*       define.argument :arg_name = pathname \
--*D*                       :type     = string
--*D*         set.description :text = "
--*D*             -- Pathname to store the account. Must be
--*D*             -- a valid pathname that is not already in use.
--*D*             -- Caller must have store rights in the referenced
--*D*             -- directory.
--*D*             "
--*D*         set.maximum_length  43
--*D*         set.mandatory
--*D*       end
--*D*
--*D*       define.argument :arg_name = init_balance \
--*D*                       :type     = integer
--*D*         set.description :text = "
--*D*             -- Initial balance of the account. Must be
--*D*             -- greater or equal to zero and less than or equal
--*D*             -- to 100000.
--*D*             "
--*D*         set.bounds        :value = 0..100000
--*D*         set.mandatory
--*D*       end
--*D*
--*D*       define.argument :arg_name = authority \
--*D*                       :type     = string
--*D*         set.description :text = "
--*D*             -- Specifies an authority list to be stored
--*D*             -- with an account. Has to be created separately
--*D*             -- invoking the manage.authority runtime command.
--*D*             -- Default value is none.
```

```
--*D*                  "
--*D*              set.maximum_length 43
--*D*              set.value_default :value = "none"
--*D*            end
--*D*          end
--*D*
--*D*
--*D*          define.command :cmd_name = store
--*D*            set.description :text = "
--*D*                -- Store an existing active account.
--*D*                -- Causes separate command set acct_cmd_store
--*D*                -- to be invoked.
--*D*                "
--*D*
--*D*            define.argument :arg_name = ref_number \
--*D*                            :type     = integer
--*D*              set.description :text = "
--*D*                  -- Reference to an account. Has to be
--*D*                  -- between 1 and 100.
--*D*                  "
--*D*              set.bounds :value = 1..100
--*D*              set.mandatory
--*D*            end
--*D*
--*D*            define.argument :arg_name = pathname \
--*D*                            :type     = string
--*D*              set.description :text = "
--*D*                  -- Pathname to store the account. Must be
--*D*                  -- a valid pathname that is not already in use.
--*D*                  -- Caller must have store rights in the referenced
--*D*                  -- directory.
--*D*                  "
--*D*              set.maximum_length   43
--*D*              set.mandatory
--*D*            end
--*D*
--*D*            define.argument :arg_name = authority \
--*D*                            :type     = string
--*D*              set.description :text = "
--*D*                  -- Specifies an authority list to be stored
--*D*                  -- with an account. Has to be created separately
--*D*                  -- invoking the manage.authority runtime command.
--*D*                  -- Default value is none.
--*D*                  "
--*D*              set.maximum_length 43
--*D*              set.value_default :value = "none"
--*D*            end
--*D*          end
--*D*
--*D*
--*D*          define.command :cmd_name = retrieve
--*D*            set.description :text = "
--*D*                -- Retrieve a stored account from a pathname
--*D*                -- and make it available for online processing.
--*D*                "
--*D*
--*D*            define.argument :arg_name = pathname \
--*D*                            :type     = string
--*D*              set.description :text = "
--*D*                  -- Pathname of a account to be retrieved. Can
--*D*                  -- be relative, absolute, or network pathname.
--*D*                  -- Must be a valid pathname and pathname must
--*D*                  -- reference an account.
--*D*                  "
--*D*              set.maximum_length :value = 43
--*D*              set.mandatory
--*D*            end
--*D*          end
--*D*
--*D*
--*D*          define.command :cmd_name = "list"
--*D*            set.description :text = "
--*D*                -- List all accounts currently available for
--*D*                -- online processing by ordinal reference number.
--*D*                "
--*D*          end
--*D*
```

```
--*D*
--*D*    define.command :cmd_name = display
--*D*      set.description :text = "
--*D*          -- Display all relevant information about an account.
--*D*          "
--*D*
--*D*      define.argument :arg_name = ref_number \
--*D*                      :type     = integer
--*D*        set.description :text = "
--*D*            -- Ordinal number referencing the account
--*D*            "
--*D*        set.bounds        :value = 0..100
--*D*        set.value_default :value = 0
--*D*      end
--*D*    end
--*D*
--*D*
--*D*    define.command :cmd_name = withdraw
--*D*      set.description :text = "
--*D*          -- Withdraw a given amount from an account
--*D*          "
--*D*
--*D*      define.argument :arg_name = ref_number \
--*D*                      :type     = integer
--*D*        set.bounds        :value = 1..100
--*D*        set.mandatory
--*D*      end
--*D*
--*D*      define.argument :arg_name = amount \
--*D*                      :type     = integer
--*D*        set.bounds        :value = 0..100000
--*D*        set.mandatory
--*D*      end
--*D*    end
--*D*
--*D*
--*D*    define.command :cmd_name = deposit
--*D*      set.description :text = "
--*D*          -- Deposit a given amount to an account
--*D*          "
--*D*
--*D*      define.argument :arg_name = ref_number \
--*D*                      :type     = integer
--*D*        set.bounds        :value = 1..100
--*D*        set.mandatory
--*D*      end
--*D*
--*D*      define.argument :arg_name = amount \
--*D*                      :type     = integer
--*D*        set.bounds        :value = 0..100000
--*D*        set.mandatory
--*D*      end
--*D*    end
--*D*
--*D*
--*D*    define.command :cmd_name = transfer
--*D*      set.description :text = "
--*D*          -- Transfer amount from source to destination.
--*D*          "
--*D*
--*D*      define.argument :arg_name = source \
--*D*                      :type     = integer
--*D*        set.bounds  :value = 1..100
--*D*        set.mandatory
--*D*      end
--*D*
--*D*      define.argument :arg_name = destination \
--*D*                      :type     = integer
--*D*        set.bounds  :value = 1..100
--*D*        set.mandatory
--*D*      end
--*D*
--*D*      define.argument :arg_name = amount \
--*D*                      :type     = integer
--*D*        set.bounds        :value = 0..100000
--*D*        set.mandatory
--*D*      end
```

```
--*D*     end
--*D*
--*D*
--*D*     define.command :cmd_name = remove
--*D*       set.description :text = "
--*D*           -- Remove an account from online processing
--*D*           -- Does not affect an accounts passive version.
--*D*           "
--*D*
--*D*       define.argument :arg_name = ref_number \
--*D*                       :type     = integer
--*D*         set.bounds        :value = 1..100
--*D*         set.mandatory
--*D*       end
--*D*     end
--*D*
--*D*
--*D*     define.command :cmd_name = destroy
--*D*       set.description :text = "
--*D*           -- Destroy an account's passive version.
--*D*           -- Does not affect an account's online representation.
--*D*           -- Fails for account's that have not been passivated.
--*D*           "
--*D*
--*D*       define.argument :arg_name = ref_number \
--*D*                       :type     = integer
--*D*         set.bounds        :value = 1..100
--*D*         set.mandatory
--*D*       end
--*D*     end
--*D*
--*D*
--*D*     define.command :cmd_name = manage.authority
--*D*       set.description :text = "
--*D*           -- Invokes the manage.authority utility to
--*D*           -- create authority list from within this
--*D*           -- program.
--*D*           "
--*D*     end
--*D*
--*D*
--*D*     define.command :cmd_name = save
--*D*       --
--*D*       -- Invokes the screensaver utility.
--*D*       --
--*D*
--*D*       define.argument :arg_name = "args" \
--*D*                       :type     = string
--*D*         set.value_default :value = ""
--*D*         set.description :text = "
--*D*             -- Arguments to pass on to screensaver
--*D*             -- Type csh command line in quotes.
--*D*             "
--*D*       end
--*D*     end
--*D*
--*D*
--*D*     define.command :cmd_name = "exit"
--*D*       set.description :text = "
--*D*           -- Exit accounting program
--*D*           "
--*D*     end
--*D*   end
--*D* exit

------------------------------------------------------------
------------------------------------------------------------
--                                                        --
--                  MESSAGE DEFINITIONS                   --
--                                                        --
------------------------------------------------------------
------------------------------------------------------------

--*D*
--*D* manage.messages
--*D*
--*D*
```

```
--*D*
--*D*   set.language :language = english
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 1 \
--*D*          :msg_name = welcome \
--*D*          :short    = "Welcome to the Account Manager"
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 2 \
--*D*          :msg_name = local_created \
--*D*          :short    = "Local account number $p1<ref_number> has \
--*D* initial balance $p2<initial_balance>."
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 3 \
--*D*          :msg_name = list_limits_exceeded \
--*D*          :short    =  \
--*D*    "You can no longer create accounts.
--*D*     Your limit of $p1<list_length_limit> has been exceeded."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 4 \
--*D*          :msg_name = unrecognized_problem \
--*D*          :short    = "An unrecognized exception has been found."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 5 \
--*D*          :msg_name = no_access \
--*D*          :short    = "You specified an invalid pathname."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 6 \
--*D*          :msg_name = invalid_account \
--*D*          :short    = "You have specified an invalid account."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 7 \
--*D*          :msg_name = directory_entry_exists \
--*D*          :short    = "You have specified an existing directory entry"
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 8 \
--*D*          :msg_name = no_default_authority \
--*D*          :short    = "There is no default authority list."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 9 \
--*D*          :msg_name = not_implemented \
--*D*          :short    = "Operation not currently implemented."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 10 \
--*D*          :msg_name = not_supported \
--*D*          :short    = "Operation not supported."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 11 \
--*D*          :msg_name = new_balance \
--*D*          :short    = "The new balance in the account
--*D*is $p1<new_balance>"
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 12 \
--*D*          :msg_name = acct_removed \
--*D*          :short    = \
```

```
--*D*     "Account with local number $p1<ref_number> has been removed."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 13 \
--*D*          :msg_name = acct_destroyed \
--*D*          :short    =  \
--*D*    "Account with pathname $p1<pathname> has been destroyed."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 14 \
--*D*          :msg_name = not_account \
--*D*          :short    =  \
--*D*    "$p1<pathname> is not an account."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 15 \
--*D*          :msg_name = not_type_rights \
--*D*          :short    =  \
--*D*    "You have insufficient rights for this account."
--*D*
--*D*
--*D*   store  :module   = 1 \
--*D*          :number   = 16 \
--*D*          :msg_name = no_master_AD \
--*D*          :short    = "This operation requires that the account \
--*D*                be stored."
--*D*
--*D*
--*D* exit

with
  Account_Mgt_Ex,
  Acct_visual,
  Acct_Types,
  Authority_List_Mgt,
  Character_Display_AM,
  Command_Execution,
  Command_Handler,
  Conversion_Support_Ex,
  Device_Defs,
  Directory_Mgt,
  Incident_Defs,
  Long_Integer_Defs,
  Message_Services,
  Passive_Store_Mgt,
  Process_Mgt,
  Process_Mgt_Types,
  System,
  System_Defs,
  System_Exceptions,
  Terminal_Defs,
  Text_Mgt,
  Transaction_Mgt,
  Unchecked_conversion,
  Window_Services;


 procedure Acct_main_loop is
   --
   -- Function:
   --   Main event loop for account managing program.
   --

     -- Variables for creating and storing accounts:
     --
     local_list:      Acct_Types.list;
       -- List of local accounts.
     list_pointer:    Acct_Types.acct_enum := Acct_Types.list_pointer_init;
       -- Pointer to first free element in "local_list".
     ref_number:      Acct_Types.acct_enum;
       -- Pointer to current element in "local_list".
     source_number:   Acct_Types.acct_enum;
     dest_number:     Acct_Types.acct_enum;
     list_exceeded:   boolean := false;
```

```
  -- True if "list" is full.
pathname:         System_Defs.text(Acct_Types.name_length_limit);
  -- Container for pathnames.
initial_balance: integer;
  -- Container for initial balances.
long_initial_balance:  Long_Integer_Defs.long_integer;
  -- Container for long integers.
amount:           Long_Integer_Defs.long_integer;
  -- Container for long integers.
new_balance:      Long_Integer_Defs.long_integer;
  -- Container for long integers.


-- Variables for Command processing:
--
input:         Device_Defs.opened_device;
  -- Opened device for top level command processing.
cmd_id:           System.short_ordinal;
  -- Ordinal identifier for commands.
cmd_name:         System_Defs.text(Acct_Types.name_length_limit);
  -- Textual identifier for commands.

-- Variables for Window output:
--
old_opened_window:    Device_Defs.opened_device;
  -- Standard input.
old_window:           Device_Defs.device;
  -- Standard input .. underlying device.
new_opened_window:    Device_Defs.opened_device;
  -- Window for display output.
new_window:           Device_Defs.device;
  -- Window for display output -- underlying device.
underlying_terminal:  Device_Defs.device;
  -- User terminal.
curr_pos:             Terminal_Defs.point_info;
  -- Current position in the opened window.
new_window_info:      Window_Services.window_style_info;
  -- Style info for new window.

-- Constants defining Window output:
--
frame_buffer:  constant Terminal_Defs.point_info :=
                        Terminal_Defs.point_info'(80, 20);
window_size:   constant Terminal_Defs.point_info :=
                        Terminal_Defs.point_info'(80, 10);
window_pos:    constant Terminal_Defs.point_info :=
                        Terminal_Defs.point_info'(1, 1);
view_pos:      constant Terminal_Defs.point_info :=
                        Terminal_Defs.point_info'(1, 1);
title_string:  constant string := "ACCOUNTS";


-- Variables for authority lists:
--
auth_list:        Authority_list_Mgt.authority_list_AD;
  -- Authority list for storing accounts.
authority_name:  System_Defs.text(Acct_Types.name_length_limit);
  -- Pathname of authority list.

-- Auxiliary variables:
--
i:             integer;
exit_status:   Incident_Defs.severity_value;
aux_text:      System_Defs.text(Window_Services.max_title);
untyped_AD:    System.untyped_word;
args:          System_Defs.text(Acct_Types.name_length_limit);
cmd_line:      System_Defs.text(Acct_Types.name_length_limit);

-- Exceptions:
--
list_exceeded_exc:      exception;
mission_accomplished:   exception;
invalid_account:        exception;
not_implemented:        exception;
new_balance_exc:        exception;
account_removed:        exception;
account_destroyed:      exception;
```

```
      not_account:            exception;

      -- Conversions:
      --
      function Untyped_from_account is new
          Unchecked_conversion(
              source => Account_Mgt_Ex.account_AD,
              target => System.untyped_word);

      function Account_from_untyped is new
          Unchecked_conversion(
              source => System.untyped_word,
              target => Account_Mgt_Ex.account_AD);


      use Incident_Defs;       -- Import some frequently used defs.
      use Long_Integer_Defs;  -- Import long integer arithmetic.


    begin
      -- Initialize account list
      --
      for i in Acct_Types.list_pointer_init .. Acct_Types.list_length_limit
          loop
        Text_Mgt.Set(
            dest   => local_list(i).name,
            source => Acct_Types.empty_text);

      end loop;

      -- Open runtime command processing:
      --
      input := Command_Handler.Open_runtime_command_processing(
          cmd_set => System_Defs.text'(14, 14, "$OEO/acct_cmds"));

      -- Open window for display output:
      --
      old_opened_window := Process_Mgt.Get_process_globals_entry(
          slot => Process_Mgt_Types.standard_input);
        -- Retrieve standard input.

      old_window := Character_Display_AM.Ops.Get_device_object(
          opened_dev => old_opened_window);
        -- Retrieve the window underlying standard input.

      underlying_terminal := Window_Services.Ops.Get_terminal(
          window => old_window);
        -- Retrieve underlying terminal.

      new_window := Window_Services.Ops.Create_window(
          terminal            => underlying_terminal,
          pixel_units         => false,
          fb_size             => frame_buffer,
          desired_window_size => window_size,
          window_pos          => window_pos,
          view_pos            => view_pos);

      Text_Mgt.Set(
          dest   => new_window_info.title,
          source => title_string);


--      Window_Services.Ops.Set_window_style(
--          window     => new_opened_window,
--          new_info   => new_window_info,
--          style_list => Window_Services.window_style_mask'
--                        (Window_Services.set_title => true,
--                         others                    => false));

      new_opened_window := Character_Display_AM.Ops.Open(
          device       => new_window,
          input_output => Device_Defs.output);

      Character_Display_AM.Ops.Clear(new_opened_window);

      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => new_opened_window,
```

```
        new_pos     => Terminal_Defs.point_info'(15,2));

    curr_pos := Terminal_Defs.point_info'(3, 5);

    Message_Services.Write_msg(
        msg_id      => Incident_Defs.incident_code'
                       (1, 1, information, System.null_word),
        no_header   => true,
        device      => new_opened_window);

    curr_pos := Terminal_Defs.point_info'(3, 2);



    loop
      begin
        -- Program block to handle exceptions.

        Command_Handler.Get_command(
            cmd_odo    => input,
            cmd_id     => cmd_id,
            cmd_name   => cmd_name);

        case cmd_id is

          -- CREATE:
          --
          when 0 =>
            -- A. Get argument from command line:
            --
            initial_balance := Command_Handler.Get_integer(
                cmd_odo     => input,
                name        => System_Defs.text'(12, 12,"init_balance"));

            -- B. Check whether there is space available:
            --
            if list_exceeded then
              -- Out of space.
              RAISE list_exceeded_exc;

            else
              -- Space available
              long_initial_balance :=
                  Long_Integer_Defs.long_integer'(0,
                  System.ordinal(initial_balance));
                -- Convert integer to long integer.

              -- C. Create account and add to local list:
              --
              local_list(list_pointer).AD :=
                  Account_Mgt_Ex.Create_account(
                      starting_balance => long_initial_balance);
              local_list(list_pointer).number := list_pointer;
              Text_Mgt.Set(
                  dest      => local_list(list_pointer).name,
                  source    => Acct_Types.local_text);
              local_list(list_pointer).stored := false;

              if list_pointer = Acct_Types.list_length_limit then
                list_exceeded := true;
                RAISE list_exceeded_exc;

              end if;
              list_pointer := list_pointer+1;
              RAISE mission_accomplished;
            end if;



          -- CSTORE:
          --
          when 1 =>
            -- A. Get arguments from command line:
            --
            initial_balance := Command_Handler.Get_integer(
                cmd_odo     => input,
```

```
      name       => System_Defs.text'(12, 12,"init_balance"));

  Command_Handler.Get_string(
      cmd_odo    => input,
      name       => System_Defs.text'(8, 8, "pathname"),
      arg_value  => pathname);

  Command_Handler.Get_string(
      cmd_odo    => input,
      name       => System_Defs.text'(9, 9, "authority"),
      arg_value  => authority_name);

  if list_exceeded then
    -- Out of space.
    RAISE list_exceeded_exc;

  else
    -- Space available
    long_initial_balance :=
        Long_Integer_Defs.long_integer'(0,
        System.ordinal(initial_balance));
      -- Convert integer to long integer.

    if Text_Mgt.Equal(authority_name, Acct_Types.none_text) then
      -- No authority list was specified. Use default.
      auth_list := null;

    else
      auth_list := Conversion_Support_Ex.
                   Authority_list_from_untyped(
                   Directory_Mgt.Retrieve(authority_name));
        -- Retrieve authority list;

    end if;

    -- C. Create account and add to local list:
    --
    local_list(list_pointer).AD     :=
        Account_Mgt_Ex.Create_stored_account(
            starting_balance => long_initial_balance,
            master           => pathname,
            authority        => auth_list);

    local_list(list_pointer).number := list_pointer;
    local_list(list_pointer).name   := pathname;
    local_list(list_pointer).stored := true;


    if list_pointer = Acct_Types.list_length_limit then
      list_exceeded := true;
      RAISE list_exceeded_exc;

    end if;

  end if;
  list_pointer := list_pointer+1;
  RAISE mission_accomplished;


-- STORE:
--
when 2 =>
  -- A. Get arguments from command line:
  --
  ref_number := Command_Handler.Get_integer(
      cmd_odo    => input,
      name       => System_Defs.text'(10, 10,"ref_number"));

  Command_Handler.Get_string(
      cmd_odo    => input,
      name       => System_Defs.text'(8, 8, "pathname"),
      arg_value  => pathname);

  Command_Handler.Get_string(
      cmd_odo    => input,
      name       => System_Defs.text'(8, 8, "pathname"),
      arg_value  => authority_name);
```

```
  if Text_Mgt.Equal(local_list(ref_number).name,
                    Acct_Types.empty_text)
      then
    -- Unassigned account.
    RAISE invalid_account;

  end if;

  if Text_Mgt.Equal(authority_name, Acct_Types.none_text) then
    -- No authority list was specified. Use default.
    auth_list := null;

  end if;

  -- Enclose passive store operations in a transaction:
  --
  Transaction_Mgt.Start_transaction;
  begin
    Directory_Mgt.Store(
        name    => pathname,
        object  => Untyped_from_account(
                       local_list(ref_number).AD),
        aut     => auth_list);

    Passive_Store_Mgt.Request_update(
        obj  => Untyped_from_account(local_list(ref_number).AD));

    Transaction_Mgt.Commit_transaction;
  exception
    when others =>
      Transaction_Mgt.Abort_transaction;
      RAISE;

  end;


  local_list(ref_number).name := pathname;
  local_list(ref_number).stored := true;



-- RETRIEVE:
--
when 3 =>
  -- A. Get arguments from command line:
  --
  Command_Handler.Get_string(
      cmd_odo     => input,
      name        => System_Defs.text'(8, 8, "pathname"),
      arg_value   => pathname);

  if list_exceeded then
    RAISE list_exceeded_exc;

  else
    -- B. Retrieve account and add to local list:
    --
    untyped_AD := Directory_Mgt.Retrieve(pathname);
    if not Account_Mgt_Ex.Is_account(untyped_AD) then
      RAISE not_account;

    end if;

    local_list(list_pointer).AD :=
        Account_from_untyped(untyped_AD);
    local_list(list_pointer).number := list_pointer;
    local_list(list_pointer).name   := pathname;
    local_list(list_pointer).stored := true;

    long_initial_balance := Account_Mgt_Ex.Get_balance(
                               local_list(list_pointer).AD);

    initial_balance := integer(long_initial_balance.l);

    if list_pointer = Acct_Types.list_length_limit then
      list_exceeded := true;
```

```
                RAISE list_exceeded_exc;

              end if;
              list_pointer := list_pointer+1;
              RAISE mission_accomplished;

          end if;

        -- LIST:
        --
        when 4 =>
          Character_Display_AM.Ops.Clear(new_opened_window);
          Acct_visual.Display_list(
              list          => local_list,
              output        => new_opened_window,
              pixel_units   => false,
              location      => curr_pos);



        -- DISPLAY:
        --
        when 5 =>
          -- A. Get arguments from command line:
          --
          ref_number := Command_Handler.Get_integer(
              cmd_odo    => input,
              name       => System_Defs.text'(10, 10,"ref_number"));

          if ref_number = 0 then
            ref_number := list_pointer-1;
          end if;

          if Text_Mgt.Equal(local_list(ref_number).name,
                            Acct_Types.empty_text)
              then
            -- Unassigned account.
            RAISE invalid_account;

          end if;

          Character_Display_AM.Ops.Clear(new_opened_window);
          Acct_visual.Display_account(
              account        => local_list(ref_number).AD,
              output         => new_opened_window,
              pixel_units    => false,
              location       => curr_pos);




        -- WITHDRAW:
        --
        when 6 =>
          -- A. Get arguments from command line:
          --
          ref_number := Command_Handler.Get_integer(
              cmd_odo    => input,
              name       => System_Defs.text'(10, 10,"ref_number"));

          initial_balance := Command_Handler.Get_integer(
              cmd_odo    => input,
              name       => System_Defs.text'(6, 6, "amount"));

          if Text_Mgt.Equal(local_list(ref_number).name,
                            Acct_Types.empty_text)
              then
            -- Unassigned account.
            RAISE invalid_account;

          end if;

          amount :=
              Long_Integer_Defs.long_integer'(0,
               System.ordinal(initial_balance));
            -- Convert integer to long integer.
```

```
            new_balance := Account_Mgt_Ex.Change_balance(
                account => local_list(ref_number).AD,
                amount  => - amount);

            RAISE new_balance_exc;


        -- DEPOSIT:
        --
        when 7 =>
            -- A. Get arguments from command line:
            --
            ref_number := Command_Handler.Get_integer(
                cmd_odo     => input,
                name        => System_Defs.text'(10, 10,"ref_number"));

            initial_balance := Command_Handler.Get_integer(
                cmd_odo     => input,
                name        => System_Defs.text'(6, 6, "amount"));

            if Text_Mgt.Equal(local_list(ref_number).name,
                              Acct_Types.empty_text)
                then
              -- Unassigned account.
              RAISE invalid_account;

            end if;

            amount :=
                Long_Integer_Defs.long_integer'(0,
                 System.ordinal(initial_balance));
              -- Convert integer to long integer.

            new_balance := Account_Mgt_Ex.Change_balance(
                account => local_list(ref_number).AD,
                amount  => amount);

            RAISE new_balance_exc;



        -- TRANSFER:
        --
        when 8 =>
            -- A. Get arguments from command line:
            --
            source_number := Command_Handler.Get_integer(
                cmd_odo     => input,
                name        => System_Defs.text'(6, 6,"source"));

            dest_number := Command_Handler.Get_integer(
                cmd_odo     => input,
                name        => System_Defs.text'(11, 11,"destination"));

            initial_balance := Command_Handler.Get_integer(
                cmd_odo     => input,
                name        => System_Defs.text'(6, 6, "amount"));

            if Text_Mgt.Equal(local_list(source_number).name,
                              Acct_Types.empty_text) or
               Text_Mgt.Equal(local_list(dest_number).name,
                              Acct_Types.empty_text)
                then
              -- Unassigned account.
              RAISE invalid_account;

            end if;

            amount :=
                Long_Integer_Defs.long_integer'(0,
                 System.ordinal(initial_balance));
              -- Convert integer to long integer.

            Account_Mgt_Ex.Transfer(
                source_account => local_list(source_number).AD,
                dest_account   => local_list(dest_number).AD,
                amount         => amount);
```

```
        -- REMOVE:
        --
        when 9 =>
          -- A. Get arguments from command line:
          --
          ref_number := Command_Handler.Get_integer(
              cmd_odo    => input,
              name       => System_Defs.text'(10, 10,"ref_number"));

          if Text_Mgt.Equal(local_list(ref_number).name,
                            Acct_Types.empty_text)
              then
            -- Unassigned account.
            RAISE invalid_account;

          end if;

          Text_Mgt.Set(
              dest   => local_list(ref_number).name,
              source => Acct_Types.empty_text);

          RAISE account_removed;


        -- DESTROY:
        --
        when 10 =>
          -- A. Get arguments from command line:
          --
          ref_number := Command_Handler.Get_integer(
              cmd_odo    => input,
              name       => System_Defs.text'(10, 10,"ref_number"));

          if Text_Mgt.Equal(local_list(ref_number).name,
                            Acct_Types.empty_text)
              then
            -- Unassigned account.
            RAISE invalid_account;

          end if;

          Account_Mgt_Ex.Destroy_account(local_list(ref_number).AD);

          Text_Mgt.Set(
              dest   => local_list(ref_number).name,
              source => Acct_Types.empty_text);

          RAISE account_destroyed;


        -- MANAGE.AUTHORITY:
        --
        when 11 =>
          exit_status := Command_Execution.Execute_command(
            command => System_Defs.text'(16, 16, "manage.authority"));

        -- SAVE:
        --
        when 12 =>
          -- A. Get arguments from command line:
          --
          Command_Handler.Get_string(
              cmd_odo    => input,
              name       => System_Defs.text'(4, 4, "args"),
              arg_value  => args);

          Text_mgt.Set(
              dest   => cmd_line,
              source => "ss ");

          Text_Mgt.Append(
              dest   => cmd_line,
              source => args);
```

```
          exit_status := Command_Execution.Execute_command(
            command => cmd_line);



        -- EXIT:
        --
        when 13 =>
          EXIT;

        when others =>
          RAISE not_implemented;

      end case;


    exception
      -- Main exception handler:

      -- LOCAL:
      --
      when account_destroyed =>
        Text_Mgt.Set(
            dest   => aux_text,
            source => pathname);
        Message_Services.Write_msg(
            msg_id    => Incident_Defs.incident_code'
                             (1, 13, information, System.null_word),
            param1    => Incident_Defs.message_parameter'
                             (txt, pathname.length, aux_text),
            no_header => true);

      when account_removed =>
        Message_Services.Write_msg(
            msg_id    => Incident_Defs.incident_code'
                             (1, 12, information, System.null_word),
            param1    => Incident_Defs.message_parameter'
                             (int, 4, integer(ref_number)),
            no_header => true);

      when invalid_account =>
        Message_Services.Write_msg(
            msg_id    => Incident_Defs.incident_code'
                             (1, 6, error, System.null_word),
            no_header => true);

      when list_exceeded_exc =>
        Message_Services.Write_msg(
            msg_id    => Incident_Defs.incident_code'
                             (1, 3, warning, System.null_word),
            param1    => Incident_Defs.message_parameter'
                             (int, 4, Acct_Types.list_length_limit),
            no_header => true);

      when mission_accomplished =>
        Message_Services.Write_msg(
            msg_id    => Incident_Defs.incident_code'
                                 (1, 2, information, System.null_word),
            param1    => Incident_Defs.message_parameter'
                             (int, 4, list_pointer-1),
            param2    => Incident_Defs.message_parameter'
                             (int, 4, initial_balance),
            no_header => true);

      when new_balance_exc =>
        Message_Services.Write_msg(
            msg_id    => Incident_Defs.incident_code'
                             (1, 11, warning, System.null_word),
            param1    => Incident_Defs.message_parameter'
                             (int, 4, integer(new_balance.l)),
            no_header => true);

      when not_account =>
        Text_Mgt.Set(
            dest   => aux_text,
            source => pathname);
        Message_Services.Write_msg(
```

```
              msg_id    => Incident_Defs.incident_code'
                           (1, 14, error, System.null_word),
              param1    => Incident_Defs.message_parameter'
                           (txt, pathname.length, aux_text),
              no_header => true);

        when not_implemented =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (1, 9, warning, System.null_word),
              no_header => true);


        -- ACCOUNT_MGT_EX:
        --
        when Account_mgt_Ex.balance_not_zero =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (0, 2, error, System.null_word),
              no_header => true);

        when Account_Mgt_Ex.insufficient_balance =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (0, 1, error, System.null_word),
              no_header => true);

        -- DIRECTORY_MGT:
        --
        when Directory_Mgt.entry_exists =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (1, 7, error, System.null_word),
              no_header => true);

        when Directory_Mgt.no_access =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (1, 5, error, System.null_word),
              no_header => true);

        when Directory_Mgt.no_default_authority_list =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (1, 8, error, System.null_word),
              no_header => true);

        -- PASSIVE_STORE_MGT:
        --
        when Passive_Store_Mgt.no_master_AD =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (1, 16, error, System.null_word),
              no_header => true);

        -- SYSTEM_EXCEPTIONS:
        --
        when System_Exceptions.insufficient_type_rights =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (1, 15, warning, System.null_word),
              no_header => true);

        when System_Exceptions.operation_not_supported =>
          Message_Services.Write_msg(
              msg_id    => Incident_Defs.incident_code'
                           (1, 10, warning, System.null_word),
              no_header => true);

--          when others =>
--            Message_Services.Write_msg(
--                msg_id    => Incident_Defs.incident_code'
--                             (1, 4, error, System.null_word),
--                no_header => true);
--            RAISE;
      end;

```

```
   end loop;

end Acct_main_loop;
```

X-A.7.2 `Acct_Visual` Package Specification

Display routines used by the account manager test driver.

```
with
  Account_Mgt_Ex,
  Acct_Types,
  Authority_List_Mgt,
  Device_Defs,
  Long_Integer_Defs,
  System_Defs,
  Terminal_Defs;

 package Acct_visual is
   --
   -- Function:
   --   This package contains procedures to display
   --   information about accounts. It is called by the
   --   Acct_main procedure.
   --
   -- Calls:
   --   o Display_account   Given an AD displays all information relevant to
   --                       the account, i. e. pathname, creator, creation,
   --                       creation time, time last read, time last modified.
   --                       and the current balance.
   --
   --   o List_account  Given a Acct_main.list, displays that list.
   --
   -- Exceptions:


   procedure Display_account(
      account:  Account_Mgt_Ex.account_AD;
        -- Account that is to be displayed.
      output:   Device_Defs.opened_device;
        -- Device to use for displaying info.
      pixel_units:  boolean := false;
        -- Whether to use character- or pixel units.
      location:     Terminal_Defs.point_info);
        -- Where to display the account.
     --
     -- Function:
     --   Displays relevant information about an account
     --   in the following format:
     --
     --   +-------------------------------------------------------------+
     --   | NAME:                  ///bla/bla/acct/bozo1                |
     --   | CREATOR:               ///bla/bla/id/bozo                   |
     --   | CREATED:               12/12/1212   15:43:59                |
     --   | LAST READ:             12/12/1212   15:43:59                |
     --   | LAST MODIFIED          12/12/1212   15:43:59                |
     --   |                                                             |
     --   |          Current Balance:   $ 146358.00                     |
     --   +-------------------------------------------------------------+
     --
     --   For accounts that have no passive version the display will
     --   look like this:
     --
     --   +-------------------------------------------------------------+
     --   | NAME:                  local                                |
     --   | CREATOR:                                                    |
     --   | CREATED:                                                    |
     --   | LAST READ:                                                  |
     --   | LAST MODIFIED:                                              |
     --   |                                                             |
     --   |          Current Balance:   $ 146358.00                     |
     --   +-------------------------------------------------------------+
     --


  procedure Display_list(
      list:         Acct_Types.list;
        -- List to display.
      output:       Device_Defs.opened_device;
        -- Device to use for displaying info.
      pixel_units:  boolean := false;
        -- Whether to use character- or pixel units.
      location:     Terminal_Defs.point_info);
```

```
          -- Where to display the list.
     --
     -- Function:
     --   Displays a list of local account in the following format:
     --
     --   <ref_number>  <stored>     <name>
     --             1   stored    ///Gemini/State/home/tobiash/savings
     --             2   local     ///Gemini/State/home/martinb/checking
     --             3   stored    ///Gemini/State/home/patty/stocks
     --
     --

 pragma external;

end Acct_visual;
```

X-A.7.3 `Acct_Visual` Package Body

Display routines used by the account manager test driver.

```
--*D*
--*D* manage.messages
--*D*
--*D*   store      :module   = 2 \
--*D*              :number   = 1 \
--*D*              :msg_name = acknowledge \
--*D*              :short    = "Type any character to continue> "
--*D* exit

with
  Account_Mgt_Ex,
  Acct_Types,
  Character_Display_AM,
  Device_Defs,
  Directory_Mgt,
  Incident_Defs,
  Long_Integer_Defs,
  Message_Services,
  Passive_Store_Mgt,
  System,
  System_Defs,
  System_Exceptions,
  Terminal_Defs,
  Text_Mgt,
  Timing_Conversions,
  Timing_String_Conversions;


 package body Acct_visual is


   procedure Display_account(
       account:     Account_Mgt_Ex.account_AD;
       output:      Device_Defs.opened_device;
       pixel_units: boolean := false;
       location: in Terminal_Defs.point_info)
     is
       account_untyped:  System.untyped_word;
         FOR account_untyped USE AT account'address;
         -- Untyped overlay.

       account_info:    Passive_Store_Mgt.passive_object_info;
       name_value:      System_Defs.text(Acct_Types.name_length_limit);
       creator_value:   System_Defs.text(Acct_Types.name_length_limit);
       created_value:   System_Defs.text(22);
       read_value:      System_Defs.text(22);
       write_value:     System_Defs.text(22);
       bal_value:       Long_Integer_Defs.string_integer;

       num_time:        Timing_conversions.numeric_time;


       no_name:    boolean := false;
       position:   Terminal_Defs.point_info;
       ID_untyped: System.untyped_word;
         FOR id_untyped USE AT account_info.owner'address;
       num_bal:    Long_Integer_Defs.long_integer;

       tb_line:  constant System_Defs.text := System_Defs.text'(65, 65,
"+---------------------------------------------------------------+");
       side:     constant System_Defs.text := System_Defs.text'(1, 1,
           "|");
       name:     constant System_Defs.text := System_Defs.text'(5, 5,
           "NAME:");
       creator:  constant System_Defs.text := System_Defs.text'(8, 8,
           "CREATOR:");
       created:  constant System_Defs.text := System_Defs.text'(8, 8,
           "CREATED:");
       read:     constant System_Defs.text := System_Defs.text'(10, 10,
           "LAST READ:");
       write:    constant System_Defs.text := System_Defs.text'(14, 14,
           "LAST MODIFIED:");
       bal:      constant System_Defs.text := System_Defs.text'(18, 18,
```

```
         "CURRENT BALANCE: $");

   begin
     -- 1. Display account template:
     --
     position := location;
     Character_Display_AM.Ops.Clear_to_bottom(output);
     Character_Display_AM.Ops.Move_cursor_absolute(
         opened_dev => output,
         new_pos    => position);
     Character_Display_AM.Ops.Write(
       -- Top line of box.
         opened_dev => output,
         buffer_VA  => tb_line.value'address,
         length     => System.ordinal(tb_line.length));

     for i in 1 .. 7 loop
       position.vert  := location.vert+i;
       position.horiz := location.horiz;
       Character_Display_AM.Ops.Move_cursor_absolute(
           opened_dev => output,
           new_pos    => position);
       Character_Display_AM.Ops.Write(
         -- Left side of box
           opened_dev => output,
           buffer_VA  => side.value'address,
           length     => System.ordinal(side.length));

       position.horiz := location.horiz+74;
       Character_Display_AM.Ops.Move_cursor_absolute(
           opened_dev => output,
           new_pos    => position);
       Character_Display_AM.Ops.Write(
         -- Right side of box.
           opened_dev => output,
           buffer_VA  => side.value'address,
           length     => System.ordinal(side.length));

     end loop;

     position.vert  := location.vert+8;
     position.horiz := location.horiz;
     Character_Display_AM.Ops.Move_cursor_absolute(
         opened_dev => output,
         new_pos    => position);
     Character_Display_AM.Ops.Write(
       -- Bottom line of box.
         opened_dev => output,
         buffer_VA  => tb_line.value'address,
         length     => System.ordinal(tb_line.length));

     position.horiz := location.horiz+1;
     position.vert  := location.vert+1;
     Character_Display_AM.Ops.Move_cursor_absolute(
         opened_dev => output,
         new_pos    => position);
     Character_Display_AM.Ops.Write(
       -- Write "NAME:" in position 2,2.
         opened_dev => output,
         buffer_VA  => name.value'address,
         length     => System.ordinal(name.length));

     position.vert  := position.vert+1;
     Character_Display_AM.Ops.Move_cursor_absolute(
         opened_dev => output,
         new_pos    => position);
     Character_Display_AM.Ops.Write(
       -- Write "CREATOR:" in position 3,2.
         opened_dev => output,
         buffer_VA  => creator.value'address,
         length     => System.ordinal(creator.length));

     position.vert  := position.vert+1;
     Character_Display_AM.Ops.Move_cursor_absolute(
         opened_dev => output,
         new_pos    => position);
     Character_Display_AM.Ops.Write(
```

```
          -- Write "CREATED:" in position 4,2.
            opened_dev => output,
            buffer_VA  => created.value'address,
            length     => System.ordinal(created.length));

        position.vert  := position.vert+1;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => position);
        Character_Display_AM.Ops.Write(
          -- Write "LAST READ:" in position 5,2.
            opened_dev => output,
            buffer_VA  => read.value'address,
            length     => System.ordinal(read.length));

        position.vert  := position.vert+1;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => position);
        Character_Display_AM.Ops.Write(
          -- Write "LAST MODIFIED:" in position 6,2.
            opened_dev => output,
            buffer_VA  => write.value'address,
            length     => System.ordinal(write.length));

        position.vert  := position.vert+2;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => position);
        Character_Display_AM.Ops.Write(
          -- Write "CURRENT BALANCE: $" in position 8,2.
            opened_dev => output,
            buffer_VA  => bal.value'address,
            length     => System.ordinal(bal.length));



        -- 2. Determine whether "account_AD" references an account
        --    with a passive version. If yes, get the account's name:
        --
        begin
          -- This block controls the scope of the exception handler.
          Directory_Mgt.Get_name(
              obj  => account_untyped,
              name => name_value);

        exception
          when Directory_Mgt.no_name =>
            Text_Mgt.Set(
                dest   => name_value,
                source => Acct_Types.local_text);
            no_name    := true;

          when others =>
            RAISE;

        end;

        --
        -- 3. Initialize values for
        --      - Creator
        --      - Creation Time
        --      - Time Last Read
        --      - Time Last Modified
        --      - Current Balance
        --    If account is unnamed initialize to "local".
        --
        if no_name then
          -- Account has no name and therefore has not
          -- been passivated.
          Text_Mgt.Set(
              dest   => creator_value,
              source => Acct_Types.local_text);
          Text_Mgt.Set(
              dest   => created_value,
              source => Acct_Types.local_text);
          Text_Mgt.Set(
```

```
            dest   => read_value,
            source => Acct_Types.local_text);
        Text_Mgt.Set(
            dest    => write_value,
            source => Acct_Types.local_text);

      else
        -- Account has a name and has been passivated.
        account_info := Passive_Store_Mgt.Request_passive_object_info(
            obj => account_untyped);

        Directory_Mgt.Get_name(
          -- Obtain user name of owner from ID.
            obj  => ID_untyped,
            name => creator_value);

        num_time := Timing_Conversions.Convert_stu_to_numeric_time(
            stu => account_info.create_time);
        Timing_string_conversions.Convert_numeric_time_to_ISO(
            num_time => num_time,
            ISO_time => created_value);

        num_time := Timing_Conversions.Convert_stu_to_numeric_time(
            stu => account_info.read_time);
        Timing_string_conversions.Convert_numeric_time_to_ISO(
            num_time => num_time,
            ISO_time => read_value);

        num_time := Timing_Conversions.Convert_stu_to_numeric_time(
            stu => account_info.write_time);
        Timing_string_conversions.Convert_numeric_time_to_ISO(
            num_time => num_time,
            ISO_time => write_value);
      end if;

      -- 4. Get balance and convert to suitable format:
      --
      num_bal := Account_Mgt_Ex.Get_balance(account);

      Long_Integer_Defs.Long_integer_image(
          number => num_bal,
          image  => bal_value);

      -- 5. Display values:
      --
      position.horiz := location.horiz+9;
      position.vert  := location.vert+1;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => output,
          new_pos    => position);
      Character_Display_AM.Ops.Write(
          opened_dev => output,
          buffer_VA  => name_value.value'address,
          length     => System.ordinal(name_value.length));

      position.horiz := location.horiz+16;
      position.vert  := position.vert+1;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => output,
          new_pos    => position);
      Character_Display_AM.Ops.Write(
          opened_dev => output,
          buffer_VA  => creator_value.value'address,
          length     => System.ordinal(creator_value.length));

      position.vert  := position.vert+1;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => output,
          new_pos    => position);
      Character_Display_AM.Ops.Write(
          opened_dev => output,
          buffer_VA  => created_value.value'address,
          length     => System.ordinal(created_value.length));

      position.vert  := position.vert+1;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => output,
```

```
            new_pos    => position);
        Character_Display_AM.Ops.Write(
            opened_dev => output,
            buffer_VA  => read_value.value'address,
            length     => System.ordinal(read_value.length));

        position.vert  := position.vert+1;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => position);
        Character_Display_AM.Ops.Write(
            opened_dev => output,
            buffer_VA  => write_value.value'address,
            length     => System.ordinal(write_value.length));

        position.vert  := position.vert+2;
        position.horiz := location.horiz+20;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => position);
        Character_Display_AM.Ops.Write(
            opened_dev => output,
            buffer_VA  => bal_value'address,
            length     => 31);

      end Display_account;



  procedure Display_list(
      list:          Acct_Types.list;
        -- List to display.
      output:        Device_Defs.opened_device;
        -- Device to use for displaying info.
      pixel_units:  boolean := false;
        -- Whether to use character- or pixel units.
      location:      Terminal_Defs.point_info)
        -- Where to display the list.

    is


    -- Auxiliary variables:
    --
    i:         integer;
    cur_pos:   Terminal_Defs.point_info;
    yes:       boolean;
    number:    System_Defs.text(5);
    step:      integer;
    act_len:   integer;



    begin
      step := 0;
      cur_pos.horiz := 1;
      cur_pos.vert := location.vert;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => output,
          new_pos    => cur_pos);
      Character_Display_AM.Ops.Clear_to_bottom(output);

      cur_pos := location;
      Character_Display_AM.Ops.Move_cursor_absolute(
          opened_dev => output,
          new_pos    => cur_pos);

      for i in Acct_Types.list_pointer_init ..
          Acct_Types.list_length_limit loop

        if not Text_Mgt.Equal(list(i).name, Acct_Types.empty_text) then
          act_len := integer'image(list(i).number)'length;

          declare
            aux_str: string(1..act_len);
          begin
            aux_str := integer'image(list(i).number);
```

```
          Text_Mgt.Set(
              dest    => number,
              source  => aux_str);
        end;

        step := step+1;
        cur_pos.vert := location.vert + (step mod 8);
        cur_pos.horiz := location.horiz+3;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => cur_pos);
        Character_Display_AM.Ops.Write(
            opened_dev => output,
            buffer_VA  => number.value'address,
            length     => System.ordinal(number.length));

        cur_pos.horiz := cur_pos.horiz+5;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => cur_pos);

        if list(i).stored then
          Character_Display_AM.Ops.Write(
              opened_dev => output,
              buffer_VA  => Acct_Types.stored_text.value'address,
              length     => System.ordinal(Acct_Types.stored_text.length));

        end if;

        cur_pos.horiz := cur_pos.horiz+Acct_Types.stored_text.length+2;
        Character_Display_AM.Ops.Move_cursor_absolute(
            opened_dev => output,
            new_pos    => cur_pos);
        Character_Display_AM.Ops.Write(
            opened_dev => output,
            buffer_VA  => list(i).name.value'address,
            length     => System.ordinal(list(i).name.length));

        if step mod 7 = 0 then
          yes := Message_Services.Acknowledge_msg(
              msg_id => Incident_Defs.incident_code'
                        (2, 1, Incident_Defs.information, System.null_word));
          cur_pos.horiz := 1;
          cur_pos.vert  := location.vert;
          Character_Display_AM.Ops.Move_cursor_absolute(
              opened_dev => output,
              new_pos    => cur_pos);
          Character_Display_AM.Ops.Clear_to_bottom(output);

        end if;

      end if;

    end loop;

  end Display_list;

end Acct_visual;
```

X-A.7.4 Account Manager Command File

Account manager command file.

```
set.program acct
create.invocation_command
  -- :set_def = acct_cmds
  --
  -- Invokes the Account Manager.
  --
end



create.runtime_command_set :cmd_def = acct_cmds \
                           :prompt  = "ACCT_MGT> "
  --
  -- Runtime commands of the account manager.
  --


  define.command :cmd_name = create
        --
        -- Create a new account with an initial balance.
        --
    define.argument :arg_name = init_balance \
                    :type     = integer
      set.bounds         :value = 0..100000
      set.mandatory
      set.description :text = "
          -- Initial balance of an account.
          -- Must be between 0 an 100000.
          "
    end
    set.description :text = "
        -- Description:
        --   Creates a local account with an initial balance.
        --   Account is not stored and will go away when program
        --   terminates unless it is stored prior to exiting.
        --
        -- Examples:
        --   *> create 10000
        --   Creates an account with an initial balance of 10000.
        --
        -- See Also:
        --
        "
  end


  define.command :cmd_name = cstore
        --
        -- Create and store a new account in one step.
        --

    define.argument :arg_name = pathname \
                    :type     = string
      set.maximum_length  43
      set.mandatory
      set.description :text = "
          -- Pathname to store the account. Must be
          -- a valid pathname that is not already in use.
          -- Caller must have store rights in the referenced
          -- directory.
          "
    end

    define.argument :arg_name = init_balance \
                    :type     = integer
      set.bounds        :value = 0..100000
      set.mandatory
      set.description :text = "
          -- Initial balance of the account. Must be
          -- greater or equal to zero and less than or equal
          -- to 100000.
          "
    end
```

```
        define.argument :arg_name = authority \
                        :type     = string
          set.maximum_length 43
          set.value_default :value = "none"
          set.description :text = "
              -- Specifies an authority list to be stored
              -- with an account. Has to be created separately
              -- invoking the manage.authority runtime command.
              -- Default value is none.
              "
        end
        set.description :text = "
            -- Description:
            --   CSTORE creates a local account with an initial balance
            --   and stores the account with a pathname. The pathname must
            --   reference an existing directory and must not already be
            --   in use. The implementation must support stored accounts,
            --   otherwise System_Exceptions.operation_not_supported will
            --   be raised.
            --
            -- Examples:
            --   *> cstore 10000 a1
            --   Creates an account called a1 with an initial balance of
            --   10000
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = store
        --
        -- Store an existing local account
        --

        define.argument :arg_name = ref_number \
                        :type     = integer
          set.description :text = "
              -- Reference to an account. Has to be
              -- between 1 and 100.
              "
          set.bounds :value = 1..100
          set.mandatory
        end

        define.argument :arg_name = pathname \
                        :type     = string
          set.description :text = "
              -- Pathname to store the account. Must be
              -- a valid pathname that is not already in use.
              -- Caller must have store rights in the referenced
              -- directory.
              "
          set.maximum_length   43
          set.mandatory
        end

        define.argument :arg_name = authority \
                        :type     = string
          set.description :text = "
              -- Specifies an authority list to be stored
              -- with an account. Has to be created separately
              -- invoking the manage.authority runtime command.
              -- Default value is none.
              "
          set.maximum_length 43
          set.value_default :value = "none"
        end
        set.description :text = "
            -- Description:
            --   Store an existing active account.
            --   The implementation must support stored accounts.
            --   Otherwise this operation will fail and the
            --   'System_Exceptions.operation' will be raised.
            --
```

```
            -- Examples:
            --    *> store :ref_number = 3 :pathname = p177
            --    Stores an account that has previously been
            --    created and assigned local number 3 with
            --    pathname 'p177'.
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = retrieve
        --
        -- Make a stored account available for processing.
        --
        define.argument :arg_name = pathname \
                        :type     = string
          set.description :text = "
              -- Pathname of a account to be retrieved. Can
              -- be relative, absolute, or network pathname.
              -- Must be a valid pathname and pathname must
              -- reference an account.
              "
          set.maximum_length :value = 43
          set.mandatory
        end
        set.description :text = "
            -- Description:
            --    Retrieve a stored account from a pathname
            --    and make it available for online processing.
            --
            -- Examples:
            --    *> retrieve  :pathname = p177
            --    Retrieves account named 'p177' in the current
            --    working directory and places it on the local list
            --    with the lowest available local number. 'pathname'
            --    must reference an account. Otherwise operation fails.
            --
            -- See Also:
            --
            "
      end



      define.command :cmd_name = "list"
        --
        -- List all accounts available for local processing.
        --
        set.description :text = "
            -- Description:
            --    List all accounts currently available for
            --    online processing by ordinal reference number.
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = display
        --
        -- Display all relevant information about an account.
        --
        define.argument :arg_name = ref_number \
                        :type     = integer
          set.description :text = "
              -- Ordinal number referencing a local account
              "
          set.bounds        :value = 0..100
          set.value_default :value = 0
        end
        set.description :text = "
            -- Description:
```

```
            --   Display all relevant information about an account.
            --   This is
            --   NAME            full network pathname.
            --   CREATOR         full name of owner.
            --   CREATED         time when created.
            --   LAST READ       time when last read.
            --   LAST MODIFIED   time when last modified.
            --   CURRENT BALANCE current balance in account.
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = withdraw
        --
        -- Withdraw amount from local account.
        --
        define.argument :arg_name = ref_number \
                        :type     = integer
          set.bounds        :value = 1..100
          set.mandatory
          set.description :text = "
              -- Reference to a local account from which
              -- 'amount' is to be withdrawn.
              "
        end

        define.argument :arg_name = amount \
                        :type     = integer
          set.bounds        :value = 0..100000
          set.mandatory
          set.description :text = "
              -- Amount to be withdrawn. Must be less than
              -- the current balance in the account.
              "
        end
        set.description :text = "
            -- Description:
            --   Withdraw a given amount from a local account.
            --   'amount' must be less than the current balance
            --   in the account. Otherwise the operation will fail.
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = deposit
        --
        -- Deposit amount in local account.
        --
        define.argument :arg_name = ref_number \
                        :type     = integer
          set.bounds        :value = 1..100
          set.mandatory
          set.description :text = "
              -- Reference to a local account in which
              -- 'amount' is to be deposited.
              "
        end

        define.argument :arg_name = amount \
                        :type     = integer
          set.bounds        :value = 0..100000
          set.mandatory
          set.description :text = "
              -- Amount to be deposited.
              "
        end
        set.description :text = "
```

```
            -- Description:
            --   Deposits a given amount in a local account.
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = transfer
        --
        -- Transfers an amount from one account to another.
        --
        define.argument :arg_name = source \
                        :type     = integer
          set.bounds  :value = 1..100
          set.mandatory
          set.description :text = "
              -- Source account for the transfer. The current
              -- balance in this account must cover the transfer.
              "
        end

        define.argument :arg_name = destination \
                        :type     = integer
          set.bounds  :value = 1..100
          set.mandatory
          set.description :text = "
              -- Destination account for the transfer.
              "
        end

        define.argument :arg_name = amount \
                        :type     = integer
          set.bounds        :value = 0..100000
          set.mandatory
          set.description :text = "
              -- Amount to be transferred from 'source' to 'dest'.
              "
        end
        set.description :text = "
            -- Description:
            --   Transfers 'amount' from 'source' to 'dest'. Transfer
            --   happens as one atomic operation in implementations that
            --   use transactions.
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = remove
        --
        -- Remove an account from the online processing.
        --
        define.argument :arg_name = ref_number \
                        :type     = integer
          set.bounds        :value = 1..100
          set.mandatory
          set.description :text = "
              -- Reference to a local account.
              "
        end
        set.description :text = "
            -- Description:
            --   Remove an account from online processing
            --   Does not affect an accounts passive version.
            --
            -- Examples:
            --
            -- See Also:
            --
```

```
            "
      end


      define.command :cmd_name = destroy
        --
        -- Destroy an account.
        --
        define.argument :arg_name = ref_number \
                        :type     = integer
          set.bounds        :value = 1..100
          set.mandatory
        end
        set.description :text = "
            -- Description:
            --   Destroys an account's passive version
            --   if the implementation supports stored accounts.
            --   Otherwise deallocates the account.
            --   A stored account still has an online version
            --   after a 'destroy'.
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end


      define.command :cmd_name = manage.authority
        --
        -- Invokes the 'manage.authority' utility.
        --
        set.description :text = "
            -- Description:
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end



      define.command :cmd_name = save
        --
        -- Invoke screensaver utility.
        --

        define.argument :arg_name = "args" \
                        :type     = string
          set.description :text = "
              -- Arguments to be passed on to
              -- screensaver utility. Type
              -- arguments exactly as you would
              -- if you invoked the screensaver
              -- from a shell, except enclose the
              -- arguments in quotes.
              "
        end
      end


      define.command :cmd_name = "exit"
        --
        -- Exits 'acct'
        set.description :text = "
            -- Description:
            --
            -- Examples:
            --
            -- See Also:
            --
            "
      end
    end
```

X-A.7.5 `Account_Types_Ex` Package Specification

```
with
  Account_Mgt_Ex,
  System_Defs;

 package Acct_Types is
   --
   -- Global type definitions and constants for accounting program.
   --
   -- Constants:
   --
   name_length_limit:   constant := 43;
   list_length_limit:   constant := 100;
   message_length:      constant := 55;
   list_pointer_init:   constant := 1;
   empty_text:          constant System_Defs.text :=
                                 System_Defs.text'(5, 5, "empty");
   none_text:           constant System_Defs.text :=
                                 System_Defs.text'(4, 4, "none");
   local_text:          constant System_Defs.text :=
                                 System_Defs.text'(5, 5, "local");
   stored_text:         constant System_Defs.text :=
                                 System_Defs.text'(6, 6, "stored");


   -- Types:
   --
   subtype acct_enum is integer range 0 .. list_length_limit;

   type local_account is
     record
       AD:      Account_Mgt_Ex.account_AD;
       number:  acct_enum;
       name:    System_Defs.text(name_length_limit);
       stored:  boolean;
     end record;

   type list is
       array(list_pointer_init .. list_pointer_init+list_length_limit-1)
       of local_account;

 end Acct_Types;
```

X-A.7.6 Account_Mgt_Ex Package Specification

Common specification for active-only, non-transaction-oriented stored, transaction-oriented stored, and distributed account type managers.

```
with Authority_List_Mgt,
     Incident_Defs,
     Long_Integer_Defs,
     Object_Mgt,
     System,
     System_Defs;

package Account_Mgt_Ex is
  --
  -- Function:
  --   Type manager for accounts.  An account
  --   contains a non-negative balance of type
  --   "Long_Integer_Defs.long_integer".
  --
  --   Several aspects of accounts are
  --   implementation-defined:
  --
  --     1. Whether accounts can be passivated.
  --
  --     2. What activation model is used for
  --        accounts.
  --
  --     3. Whether account operations are
  --        atomic, either succeeding completely
  --        or failing completely.
  --
  --     4. Whether an account object can
  --        simultaneously be used by multiple
  --        processes within a single job.
  --
  --     5. Whether the account manager is
  --        distributed, providing service at
  --        at multiple nodes in a distributed
  --        system, regardless of which nodes
  --        accounts are stored at.
  --
  --     6. Some of the protection provided
  --        between the account manager and other
  --        services.
  --
  --     7. How and where the account TDO is defined
  --        (so long as its lifetime is >= the lifetime
  --        of any account).
  --
  --     8. Account attributes.
  --
  --     9. Account manager initialization requirements.
  --
  -- Calls:
  --   Is_account            - Checks whether an AD
  --                           references an account.
  --
  --   Create_account        - Creates an account
  --                           with an initial balance.
  --
  --   Create_stored_account - Creates and stores an account.
  --
  --   Get_balance           - Returns an account's
  --                           balance.
  --
  --   Change_balance        - Changes an account's
  --                           balance.
  --
  --   Transfer              - Moves an amount between
  --                           accounts.
  --
  --   Destroy_account       - Destroys an account.
  --
  -- Messages:

  insufficient_balance_code:
```

```
        constant Incident_Defs.incident_code :=
        (0, 1, Incident_Defs.error, System.null_word);

   --*D* manage.messages
   --*D* store :module=0 :number=1 \
   --*D* :msg_name=insufficient_balance_code \
   --*D* :short= \
   --*D* "An account operation failed because it\
   --*D* would create a negative balance."

   balance_not_zero_code:
        constant Incident_Defs.incident_code :=
        (0, 2, Incident_Defs.error, System.null_word);

   --*D* store :module=0 :number=2 \
   --*D* :short= \
   --*D* "An account cannot be destroyed because\
   --*D* it has a non-zero balance."
   --*D* exit

   -- Exceptions:
   --
   insufficient_balance:  exception;
     pragma exception_value(insufficient_balance,
         insufficient_balance_code'address);
     -- An operation failed because it would
     -- cause a negative account balance.

   balance_not_zero:      exception;
     pragma exception_value(balance_not_zero,
         balance_not_zero_code'address);
     -- "Destroy_account" was called on an account
     -- with a nonzero balance.

--
-- History:
--   11-01-1985:  Martin L.  Buchanan, Initial version.
--   04-04-1988:  Tobias Haas
--                  Revised in order to unify all
--                  account manager examples.
--

   type account_object is limited private;

   type account_AD is access account_object;
     pragma access_kind(account_AD, AD);
     -- User view of an account.


   change_rights:   constant
        Object_Mgt.rights_mask :=
        Object_Mgt.modify_rights;
     -- Required to change an account's balance.

   destroy_rights:  constant
        Object_Mgt.rights_mask :=
        Object_Mgt.control_rights;
     -- Required to destroy an account.


   function Is_account(
        obj:  System.untyped_word)  -- AD to check.
     return boolean;
       -- true if "obj" references an account,
       -- else false.
     pragma protected_return(Is_account);
       --
       -- Function:
       --   Checks whether "obj" references an
       --   account.



   function Create_account(
        starting_balance:
            Long_Integer_Defs.long_integer :=
            Long_Integer_Defs.zero)
```

```
          -- Initial balance of the account.
    return account_AD;
      -- New account with all type rights and no
      -- rep rights.
    pragma protected_return(Create_account);
    --
    -- Function:
    --   Creates an account and returns an AD with all
    --   type rights. The caller is responsible for
    --   storing the AD and updating the object.
    --
    --   "starting_balance" must be nonnegative.
    --
    -- Exceptions:
    --   insufficient_balance:
    --     A negative balance was supplied.
    --
    --   Passive_Store_Mgt.no_master_AD:
    --     The object provided to store the AD in, has
    --     no master AD.


  function Create_stored_account(
      starting_balance:
          Long_Integer_Defs.long_integer :=
          Long_Integer_Defs.zero;
        -- Initial balance of the account.
      master:  System_Defs.text;
        -- Text record that holds the pathname
        -- for the master AD.
      authority:
          Authority_List_Mgt.authority_list_AD :=
          null)
        -- Optional authority list.
    return account_AD;
      -- AD to the account with all type rights and no
      -- rep rights.
    pragma protected_return(Create_stored_account);
    --
    -- Function:
    --   Creates a new account and stores the master AD
    --   under the pathname given by "master".
    --   Caller must have store rights for the named
    --   directory.
    --   The pathname cannot already be in use.
    --   "starting_balance" must be nonnegative.
    --
    --   If "authority" is null, then the new account's
    --   authority list will be either (in that order) the
    --   containing directory's default authority list, if
    --   there is one, or the caller's default authority list.
    --   If none of these three is available,
    --   "Directory_Mgt.no_default_authority_list" will be
    --   raised.
    --
    -- Exceptions:
    --   insufficient_balance:
    --     A negative starting balance was supplied.
    --
    --   Directory_Mgt.entry_exists:
    --     The pathname provided is already in use.
    --
    --   Direcotry_Mgt.no_default_authority_list:
    --     No authority list was specified, the target
    --     directory has no default authority list and there
    --     is no default authority list in the caller's
    --     process globals.


  function Get_balance(
      account:  account_AD)
        -- Any account.
    return Long_Integer_Defs.long_integer;
      -- Current balance.
    pragma protected_return(Get_balance);
    --
    -- Function:
```

```
  --   Returns an account's current balance.


 function Change_balance(
   account:  account_AD;
    -- Account with change rights.
   amount:   Long_Integer_Defs.long_integer)
    -- Amount added to balance.
  return Long_Integer_Defs.long_integer;
   -- New balance, equal to old balance
   -- plus "amount".
  pragma protected_return(Change_balance);
  --
  -- Function:
  --   Adds "amount" to an account's balance
  --   and returns the new balance.  The new
  --   balance cannot be negative.
  --
  -- Exceptions:
  --   insufficient_balance


 procedure Transfer(
   source_account:  account_AD;
    -- Account with change rights.
   dest_account:  account_AD;
    -- Account with change rights.
   amount:   Long_Integer_Defs.long_integer);
    -- Amount transferred from source to
    -- destination accounts; it can be
    -- positive or negative.  Cannot cause
    -- a negative balance in either account.
  pragma protected_return(Transfer);
  --
  -- Function:
  --   Subtracts "amount" from "source_account"
  --   and adds "amount" to "dest_account".
  --
  -- Exceptions:
  --   insufficient_balance


 procedure Destroy_account(
   account:  account_AD);
    -- Account with destroy rights.  The
    -- account's balance must be zero.
  pragma protected_return(Destroy_account);
  --
  -- Function:
  --   Destroys an account.
  --
  --   The passive version, caller's active version,
  --   and any master directory entry are destroyed.
  --
  -- Notes:
  --   Any subsequent "Get_balance",
  --   "Change_balance", or "Transfer" call
  --   will raise "object_has_no_representation"
  --   in the "System_Exceptions" package.
  --
  -- Exceptions:
  --   balance_not_zero


 pragma external;
  -- Required if this package is used with the "virtual"
  -- compilation model, which supports multiple domains
  -- and multiple subsystems.

private

 type account_object is
  -- Empty dummy record.  The real object
  -- format is defined in the package body.
  record
   null;
  end record;
```

```
end Account_Mgt_Ex;
```

X-A.7.7 Account_Mgt_Ex (Active Only) Package Body

Active-only package implementation of the account type manager.

```
with Access_Mgt,
     Attribute_Mgt,
     Long_Integer_Defs,
     Object_Mgt,
     Passive_Store_Mgt,
     System_Defs,
     System_Exceptions;

package body Account_Mgt_Ex is
  --
  -- Logic:
  --   This is an 'active-only' implementation of
  --   the account manager, with these characteristics:
  --     1. Accounts cannot be passivated.
  --
  --     2. Account operations are atomic.
  --
  --     3. An account should not be concurrently
  --        used by more than one process in a
  --        single job.
  --
  --     4. Accounts and the account TDO are local
  --        to the job that uses them.
  --
  --     5. The account TDO has the passive store
  --        attribute.
  --
  --     6. Initialization of the account manager
  --        is done within each job that uses it.
  --        Initialization creates the account TDO
  --        and assigns the passive store attribute
  --        so that accounts are active-only.
  --

  use Long_Integer_Defs;
    -- Import "long_integer" operators.

  type account_rep_object is
    record
      balance:  Long_Integer_Defs.long_integer;
        -- Current balance.
    end record;

  type account_rep_AD is access account_rep_object;
    pragma access_kind(account_rep_AD, AD);
    -- Private view of an account.

  account_TDO:  constant Object_Mgt.TDO_AD :=
                         Object_Mgt.Create_TDO;
    -- This declaration is elaborated each time
    -- this package is initialized, that is, each
    -- time a job using the package runs.  This
    -- technique for creating a TDO is only useful
    -- for objects that are completely local to
    -- a job and never stored or otherwise exported
    -- outside the creating job.


  function Is_account(
      obj:  System.untyped_word)
    return boolean
    --
    -- Logic:
    --   If "obj" is not null, retrieve the object's
    --   TDO and check whether it is the account TDO.
  is
    use Object_Mgt, System;
      -- Import "=" for "Object_Mgt.TDO_AD" and
      -- "System.untyped_word".
  begin
    return obj /= System.null_word and then
           Object_Mgt.Retrieve_TDO(obj) = account_TDO;
  end Is_account;
```

```
function Create_account(
    starting_balance:
        Long_Integer_Defs.long_integer :=
        Long_Integer_Defs.zero)
  return account_AD
  --
  -- Logic:
  --   1. Checks starting balance.
  --   2. Allocates an account.
  --   3. Initialize balance field,
  --   4. Remove rep rights on the returned AD.
is
  account:  account_AD;
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;
    -- One word viewed with three Ada types.
begin
  if starting_balance < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    account_untyped := Object_Mgt.Allocate(
        size => Object_Mgt.object_size(
                (account_rep_object'size + 31)/32),
          -- Expression computes number of words
          -- required to hold the number of bits
          -- in an account.
        tdo  => account_TDO);

    account_rep.all := account_rep_object'(
        balance => starting_balance);

    account_untyped := Access_Mgt.Remove(
        AD     => account_untyped,
        rights => Object_Mgt.read_write_rights);
    RETURN account;

  end if;
end Create_account;


function Create_stored_account(
    starting_balance:
        Long_Integer_Defs.long_integer :=
        Long_Integer_Defs.zero;
    master:  System_Defs.text;
    authority:
        Authority_List_Mgt.authority_list_AD := null)
  return account_AD
  --
  -- Logic:
  --   This call is  not supported by this implementation.
  --
is
begin
  RAISE System_Exceptions.operation_not_supported;
  RETURN null;

end Create_stored_account;


function Get_balance(
    account:  account_AD)
  return Long_Integer_Defs.long_integer
  --
  -- Logic:
  --   Amplifies read rights on "account" and
  --   returns the balance field.
is
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;
```

```
begin
  account_untyped := Access_Mgt.Amplify(
      AD     => account_untyped,
      rights => Object_Mgt.read_rights,
      tdo    => account_TDO);
  return account_rep.balance;
end Get_balance;


function Change_balance(
    account:  account_AD;
    amount:   Long_Integer_Defs.long_integer)
  return Long_Integer_Defs.long_integer
  --
  -- Logic:
  --   1. Imports rep rights on account if account
  --      has change rights.
  --   2. Adds "amount" to the existing balance to
  --      compute the prospective new balance.
  --      "amount" can be positive (a deposit),
  --      negative (a withdrawal), or zero.
  --   3. If new balance would be negative, raises
  --      "insufficient_balance" and does not change
  --      the balance.
  --   4. If new balance would be positive, then
  --      stores the new balance and also returns it.
  --   5. Makes the update an atomic operation. If anything
  --      goes wrong the update is rolled back.
is
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;
  new_balance:  Long_Integer_Defs.long_integer;
    -- Holds the new balance until a decision is
    -- made whether to store it in the account.
  old_balance:  Long_Integer_Defs.long_integer;
    -- Holds the old balance in case the operation
    -- has to be rolled back.
begin
  account_untyped := Access_Mgt.Import(
      AD     => account_untyped,
      rights => change_rights,
      tdo    => account_TDO);

  new_balance := account_rep.balance + amount;

  if new_balance < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    begin
      old_balance := account_rep.balance;
      account_rep.balance := new_balance;
      RETURN new_balance;
    exception
      -- An exception in this inner block means
      -- that something has gone wrong with the
      -- update. The old balance is restored.
      when others =>
        account_rep.balance := old_balance;
      RAISE;
    end;

  end if;
end Change_balance;


procedure Transfer(
    source_account:  account_AD;
    dest_account:    account_AD;
    amount:          Long_Integer_Defs.long_integer)
  --
  -- Logic:
  --   1. Imports rep rights on both accounts if
  --      they have change rights.
  --   2. Compute the prospective new balances,
```

```
  --        by subtracting "amount" from the source
  --        account's balance and adding it to the
  --        destination account's balance.
  --        "amount" can be positive, negative,
  --        or zero.
  --     3. If either new balance would be negative,
  --        raises "insufficient_balance" and does
  --        not change the balance.
  --     4. Assigns the new balances.  If an
  --        exception occurs between assigning the
  --        new source balance and the new destination
  --        balance, a handler rolls back the source
  --        balance to its old value, preserving
  --        atomicity.
is
  source_rep:  account_rep_AD;
  FOR source_rep USE AT source_account'address;
  source_untyped:  System.untyped_word;
  FOR source_untyped USE AT source_account'address;
  old_source_bal:  Long_Integer_Defs.long_integer;
    -- Used to remember the old source balance in case
    -- it needs to be restored if an exception occurs.
  new_source_bal:  Long_Integer_Defs.long_integer;
    -- Holds the new source balance until a decision is
    -- made whether to store it in the account.

  dest_rep:  account_rep_AD;
  FOR dest_rep USE AT dest_account'address;
  dest_untyped:  System.untyped_word;
  FOR dest_untyped USE AT dest_account'address;
  old_dest_bal:  Long_Integer_Defs.long_integer;
    -- Used to remember the old destination balance in case
    -- it needs to be restored if an exception occurs.
  new_dest_bal:  Long_Integer_Defs.long_integer;
    -- Holds the new destination balance until a decision
    -- is made whether to store it in the account.

begin
  source_untyped := Access_Mgt.Import(
      AD     => source_untyped,
      rights => change_rights,
      tdo    => account_TDO);
  dest_untyped := Access_Mgt.Import(
      AD     => dest_untyped,
      rights => change_rights,
      tdo    => account_TDO);

  new_source_bal := source_rep.balance - amount;
  new_dest_bal := dest_rep.balance + amount;

  if new_source_bal < Long_Integer_Defs.zero
     or else
     new_dest_bal < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    old_source_bal := source_rep.balance;
    old_dest_bal := dest_rep.balance;
    -- Old balances are recorded here
    -- in case the update will have to be
    -- rolled back.
    begin
      source_rep.balance := new_source_bal;
      dest_rep.balance := new_dest_bal;
    exception
      -- An exception in this inner block means
      -- that something has gone wrong with
      -- the update. Restore the old balances to make
      -- this operation atomic, then
      -- reraise the exception.
      when others =>
        source_rep.balance := old_source_bal;
        dest_rep.balance := old_dest_bal;
        RAISE;

    end;
    RETURN;
```

```
    end if;
  end Transfer;


  procedure Destroy_account(
      account:  account_AD)
    --
    -- Logic:
    --   Imports rep rights on account if account
    --   has destroy rights.

    --   If account's balance is not zero, raises
    --   "balance_not_zero".
    --
    --   Otherwise, destroys the account.
  is
    account_rep:  account_rep_AD;
    FOR account_rep USE AT account'address;
    account_untyped:  System.untyped_word;
    FOR account_untyped USE AT account'address;
  begin
    account_untyped :=  Access_Mgt.Import(
          AD     => account_untyped,
          rights => destroy_rights,
          tdo    => account_TDO);

    if account_rep.balance /= Long_Integer_Defs.zero then
      RAISE balance_not_zero;

    else
      Object_Mgt.Deallocate(account_untyped);

    end if;
  end Destroy_account;


begin
  declare
    passive_store_impl:  constant
        Passive_Store_Mgt.PSM_attributes_AD := new
        Passive_Store_Mgt.PSM_attributes_object;
    passive_store_impl_untyped:  System.untyped_word;
    FOR passive_store_impl_untyped USE AT
        passive_store_impl'address;
  begin
    Passive_Store_Mgt.Set_refuse_filters(
        passive_store_impl);
    Attribute_Mgt.Store_attribute_for_type(
        tdo       => account_TDO,
        attr_ID   => Passive_Store_Mgt.PSM_attributes_ID,
        attr_impl => passive_store_impl_untyped);
  end;
end Account_Mgt_Ex;
```

X-A.7.8 `Account_Mgt_Ex` (Stored, Non-transaction-oriented) Package Body

Non-transaction-oriented implementation of the type manager for stored accounts.

```
with Access_Mgt,
     Authority_List_Mgt,
     Directory_Mgt,
     Long_Integer_Defs,
     Object_Mgt,
     Passive_Store_Mgt,
     System,
     System_Defs;

package body Account_Mgt_Ex is
  --
  -- Logic:
  --   This is an implementation of the
  --   account manager with these characteristics:
  --
  --     * Operations are NOT guaranteed to be
  --       transaction-oriented or atomic.
  --
  --     * An account should NOT be concurrently
  --       used, not by concurrent jobs and not by
  --       concurrent processes in the same job.
  --
  --     * The account TDO must already exist in
  --       the distributed system's directory structure.
  --       The "bind" pragma is used to bind to the
  --       stored TDO.
  --
  --     * The multiple activation model is used.


  use Long_Integer_Defs,  -- Import long integer
                          -- operators.
      System;             -- Import ordinal operators.


  type account_rep_object is
    record
      balance:  Long_Integer_Defs.long_integer;
        -- Current balance.
    end record;

  type account_rep_AD is access account_rep_object;
    pragma access_kind(account_rep_AD, AD);
    -- Private view of an account.


  account_TDO:  constant Object_Mgt.TDO_AD := null;
    -- This is a constant AD but not really null; its
    -- filled in with an AD retrieved by the linker.
    pragma bind(account_TDO,
                "account");
      -- Bind to TDO for accounts.


  function Is_account(
      obj:  System.untyped_word)
    return boolean
    --
    -- Logic:
    --   If "obj" is not null, retrieve the object's
    --   TDO and check whether it is the account's TDO.
  is
    use Object_Mgt;  -- Import "=" for type "TDO_AD".
  begin
    return obj /= System.null_word
           and then
           Object_Mgt.Retrieve_TDO(obj) = account_TDO;
  end Is_account;


  function Create_account(
      starting_balance:
          Long_Integer_Defs.long_integer :=
```

```
          Long_Integer_Defs.zero)
    return account_AD
    --
    -- Logic:
    --   1. Check the initial balance.
    --
    --   2. Allocate and initialize the account object.
    --
    --   3. Remove rep rights for the exported AD.
    --      The caller is responsible for storing
    --      the AD and updating the object.
    --
    --   4. Return the AD without rep rights.
  is
    account:            account_AD;
    account_untyped:  System.untyped_word;
    FOR account_untyped USE AT account'address;
      -- Account with no rep rights, viewed with
      -- either of two types.

    account_rep:      account_rep_AD;
    account_rep_untyped:  System.untyped_word;
    FOR account_rep_untyped USE AT
        account_rep'address;
      -- Account with rep rights, viewed with
      -- either of two types.

  begin
    -- 1. Check the initial balance:
    --
    if starting_balance < Long_Integer_Defs.zero then
      RAISE insufficient_balance;

    else
      -- 2. Allocate and initialize the account object:
      --
      account_rep_untyped := Object_Mgt.Allocate(
          size => (account_rep_object'size + 31)/32,
          tdo  => account_TDO);
      account_rep.all := account_rep_object'(
          balance => starting_balance);

      -- 3. Remove rep rights for the exported AD:
      --
      account_untyped := Access_Mgt.Remove(
          AD     => account_rep_untyped,
          rights => Object_Mgt.read_write_rights);

      -- 4. Return the account AD with no rep rights:
      --
      RETURN account;

    end if;

  end Create_account;



  function Create_stored_account(
      starting_balance:
          Long_Integer_Defs.long_integer :=
          Long_Integer_Defs.zero;
      master:     System_Defs.text;
      authority:
          Authority_List_Mgt.authority_list_AD := null)
    return account_AD
    --
    -- Logic:
    --   1. Check the initial balance.
    --
    --   2. Allocate and initialize the account object.
    --
    --   3. Remove rep rights for the exported and master
    --      AD.
    --
    --   4. Store the master AD.
    --      Use "authority" as authority list to store the
```

```
    --       account. If "authority" is null, the default
    --       authority list of the target directory is used.
    --       If there is none the caller's authority list in
    --       the process globals is used.
    --
    --   5. Passivate the account object itself.
    --
    --   6. Return the AD without rep rights.
  is
    account:            account_AD;
    account_untyped:  System.untyped_word;
    FOR account_untyped USE AT account'address;
      -- Account with no rep rights, viewed with
      -- either of two types.

    account_rep:      account_rep_AD;
    account_rep_untyped:  System.untyped_word;
    FOR account_rep_untyped USE AT
        account_rep'address;
      -- Account with rep rights, viewed with
      -- either of two types.

  begin
    -- 1. Check the initial balance:
    --
    if starting_balance < Long_Integer_Defs.zero then
      RAISE insufficient_balance;

    else
      -- 2. Allocate and initialize the account object:
      --
      account_rep_untyped := Object_Mgt.Allocate(
          size => (account_rep_object'size + 31)/32,
          tdo  => account_TDO);
      account_rep.all := account_rep_object'(
          balance => starting_balance);

      -- 3. Remove rep rights for the exported and
      --    master AD:
      --
      account_untyped := Access_Mgt.Remove(
          AD     => account_rep_untyped,
          rights => Object_Mgt.read_write_rights);

      -- 4. Store the master AD:
      --
      Directory_Mgt.Store(
          name    => master,
          object  => account_untyped,
          aut     => authority);

      -- 5. Passivate the account object itself:
      --
      Passive_Store_Mgt.Update(account_rep_untyped);

      -- 6. Return the account AD with no rep rights:
      --
      RETURN account;

    end if;
  end Create_stored_account;


  function Get_balance(
      account:  account_AD)
    return Long_Integer_Defs.long_integer
    --
    -- Logic:
    --   1. Amplify rep rights on the account AD.
    --
    --   2. Return the balance.
  is
    account_rep:  account_rep_AD;
    FOR account_rep USE AT account'address;
    account_untyped:  System.untyped_word;
    FOR account_untyped USE AT account'address;

```

```
begin
  account_untyped := Access_Mgt.Amplify(
      AD     => account_untyped,
      rights => Object_Mgt.read_write_rights,
      tdo    => account_TDO);

  return account_rep.balance;
end Get_balance;


function Change_balance(
    account:  account_AD;
    amount:   Long_Integer_Defs.long_integer)
  return Long_Integer_Defs.long_integer
  --
  -- Logic:
  --   1. Import the account AD, checking for
  --      change rights and adding rep rights.
  --
  --   2. If the new balance would be negative,
  --      then exit with an exception.
  --
  --   3. Otherwise, change the balance, update
  --      the passive version, and return the
  --      new balance.
is
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;
begin
  account_untyped := Access_Mgt.Import(
          AD     => account_untyped,
          rights => change_rights,
          tdo    => account_TDO);
  if account_rep.balance + amount < zero then
    RAISE insufficient_balance;

  else
    account_rep.balance :=
        account_rep.balance + amount;
    Passive_Store_Mgt.Update(account_untyped);
    RETURN account_rep.balance;

  end if;
end Change_balance;


procedure Transfer(
    source_account:  account_AD;
    dest_account:    account_AD;
    amount:  Long_Integer_Defs.long_integer)
  --
  -- Logic:
  --   1. Import the account ADs, checking for
  --      change rights and adding rep rights.
  --
  --   2. If either new balance would be negative,
  --      then exit with an exception.
  --
  --   3. Otherwise, change the balances, update
  --      the passive versions, and return.
  --
  -- Warning:
  --   This implementation is not atomic; a change
  --   may be made in the source account but not
  --   in the destination account if an exception,
  --   system crash, or other error intervenes.
is
  source_rep:  account_rep_AD;
  FOR source_rep USE AT source_account'address;
  source_untyped:  System.untyped_word;
  FOR source_untyped USE AT source_account'address;

  dest_rep:  account_rep_AD;
  FOR dest_rep USE AT dest_account'address;
  dest_untyped:  System.untyped_word;
```

```
  FOR dest_untyped USE at dest_account'address;
begin
  source_untyped := Access_Mgt.Import(
      AD     => source_untyped,
      rights => change_rights,
      tdo    => account_TDO);
  dest_untyped := Access_Mgt.Import(
      AD     => dest_untyped,
      rights => change_rights,
      tdo    => account_TDO);

  if source_rep.balance - amount < zero
      or else
      dest_rep.balance + amount < zero
      then
    RAISE insufficient_balance;

  else
    source_rep.balance :=
        source_rep.balance - amount;
    dest_rep.balance   :=
        dest_rep.balance + amount;
    Passive_Store_Mgt.Update(source_untyped);
    Passive_Store_Mgt.Update(dest_untyped);
    RETURN;

  end if;
end Transfer;


procedure Destroy_account(
    account:  account_AD)
  --
  -- Logic:
  --   1. Import the account AD, checking for
  --      destroy rights and amplifying rep rights.
  --
  --   2. Check that the account's balance is zero.
  --      If it isn't, raise an exception.  If it
  --      is, execute the remaining steps.
  --
  --   3. Destroy the account's passive version.
  --
  --   4. Get the name of the account's master
  --      directory entry (if any).  Delete that
  --      directory entry.  Note that other
  --      entries and even a master AD may remain
  --      for the account.
  --
  --   5. Deallocate the account's active version.
is
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;

  path_length:  integer := 60;
    -- Initial text length for name assigned
    -- by "Directory_Mgt.Get_name".  If
    -- insufficient, then the value is
    -- increased and the operation is
    -- repeated.
begin
  account_untyped := Access_Mgt.Import(
      AD     => account_untyped,
      rights => destroy_rights,
      tdo    => account_TDO);

  if account_rep.balance /=
      Long_Integer_Defs.zero then
    RAISE balance_not_zero;

  else
    Passive_Store_Mgt.Destroy(account_untyped);

    loop
      declare
```

```
          path_text:  System_Defs.text(path_length);
        begin
          Directory_Mgt.Get_name(
              obj  => account_untyped,
              name => path_text);  -- out.
          if path_text.length >
             path_text.max_length then
            -- Text was lost.  Retry:
            path_length := path_text.length;
          else
            Directory_Mgt.Delete(path_text);
            EXIT;

          end if;
        exception
          when Directory_Mgt.no_name =>
            EXIT;

        end;
      end loop;

      Object_Mgt.Deallocate(account_untyped);
    end if;
  end Destroy_account;

end Account_Mgt_Ex;
```

X-A.7.9 `Account_Mgt_Ex` (Stored, Transaction-oriented) Package Body

Transaction-oriented implementation of the type manager for stored accounts.

```
with Access_Mgt,
     Authority_List_Mgt,
     Directory_Mgt,
     Long_Integer_Defs,
     Object_Mgt,
     Passive_Store_Mgt,
     System,
     System_Defs,
     System_Exceptions,
     Transaction_Mgt;

package body Account_Mgt_Ex is
  --
  -- Logic:
  --   This is an implementation of the
  --   account manager with these characteristics:
  --
  --     * All operations are transaction-oriented,
  --       participating in any default transaction
  --       or else creating a transaction for the
  --       duration of the operation.
  --
  --     * An account should not be concurrently
  --       used by more than one process in a single
  --       job, unless an external locking protocol
  --       is used.
  --
  --     * The account TDO must already exist in
  --       the distributed system's directory structure.
  --       The "bind" pragma is used to bind to the
  --       stored TDO.
  --
  --     * The multiple activation model is used.


  use Long_Integer_Defs,   -- Import "long_integer", "zero",
                           -- arithmetic and relational operators.
      System,              -- Import ordinal operators.
      Transaction_Mgt;     -- Import transaction calls.


  type account_rep_object is
    record
      balance:  Long_Integer_Defs.long_integer;
        -- Current balance.
    end record;

  type account_rep_AD is access account_rep_object;
    pragma access_kind(account_rep_AD, AD);
    -- Private view of an account.

  account_TDO:  constant Object_Mgt.TDO_AD := null;
    -- This is a constant AD but not really null; its
    -- filled in with an AD retrieved by the linker.
    pragma bind(account_TDO,
                "account");
      -- Bind to TDO for accounts.


  function Is_account(
      obj:  System.untyped_word)
    return boolean
    --
    -- Logic:
    --   If "obj" is not null, retrieve the object's
    --   TDO and check whether it is the account's TDO.
  is
    use Object_Mgt;  -- Import "=" for type "TDO_AD".
  begin
    return obj /= System.null_word
           and then
           Object_Mgt.Retrieve_TDO(obj) = account_TDO;
  end Is_account;
```

```


function Create_account(
    starting_balance:
        Long_Integer_Defs.long_integer :=
        Long_Integer_Defs.zero)
  return account_AD
  --
  -- Logic:
  --   1. Check the initial balance.
  --
  --   2. Allocate and initialize the account object.
  --
  --   3. Return AD with no rep rights.
  --
  --   4. If any exception occurs, abort any local
  --      transaction, deallocate the account,
  --      and reraise the exception.
  --
is
  account:           account_AD;
  account_untyped:   System.untyped_word;
  FOR account_untyped USE AT account'address;
    -- Account with no rep rights, viewed with
    -- either of two types.

  account_rep:       account_rep_AD;
  account_rep_untyped:  System.untyped_word;
  FOR account_rep_untyped USE AT
      account_rep'address;
    -- Account with rep rights, viewed with
    -- either of two types.

begin
  -- 1. Check the initial balance:
  --
  if starting_balance < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    -- 2. Allocate and initialize the account object:
    --
    account_rep_untyped := Object_Mgt.Allocate(
        size => (account_rep_object'size + 31)/32,
        tdo  => account_TDO);
    begin
      -- Inside this block it is guaranteed
      -- that the object has been allocated.
      account_rep.all := account_rep_object'(
          balance => starting_balance);

      -- 3. Remove rep rights for the exported AD:
      --
      account_untyped := Access_Mgt.Remove(
          AD     => account_rep_untyped,
          rights => Object_Mgt.read_write_rights);

    exception
      -- 4. If any exception occurs, abort any local
      --    transaction, deallocate the account,
      --    and reraise the exception:
      --
      when others =>
        Object_Mgt.Deallocate(account_untyped);
        RAISE;

    end;

    RETURN account;

  end if;
end Create_account;

function Create_stored_account(
    starting_balance:
        Long_Integer_Defs.long_integer :=
        Long_Integer_Defs.zero;
```

```
      master: System_Defs.text;
      authority:
          Authority_List_Mgt.authority_list_AD := null)
    return account_AD
    --
    -- Logic:
    --   1. Check the initial balance.
    --
    --   2. Allocate and initialize the account object.
    --
    --   3. Remove rep rights for the exported and
    --      master AD.
    --
    --   4. Start a local transaction if there is
    --      not a transaction on the stack.
    --
    --   5. Store the master AD.
    --      Use "authority" as authority list to store the
    --      account. If no authority list has be explicitly
    --      specified the default authority of the target
    --      directory  is used. If there is none the caller's
    --      authority list in the process globals is used instead.
    --
    --   6. Passivate the account object itself.
    --
    --   7. Commit any local transaction.
    --
    --   8. If any exception occurs, abort any local
    --      transaction, deallocate the account,
    --      and reraise the exception.
  is
    account:           account_AD;
    account_untyped:  System.untyped_word;
    FOR account_untyped USE AT account'address;
      -- Account with no rep rights, viewed with
      -- either of two types.

    account_rep:       account_rep_AD;
    account_rep_untyped:  System.untyped_word;
    FOR account_rep_untyped USE AT
        account_rep'address;
      -- Account with rep rights, viewed with
      -- either of two types.

    trans:  boolean := false;
      -- True if a local transaction is started.
  begin
    -- 1. Check the initial balance:
    --
    if starting_balance < Long_Integer_Defs.zero then
      RAISE insufficient_balance;

    else
      -- 2. Allocate and initialize the account object:
      --
      account_rep_untyped := Object_Mgt.Allocate(
          size => (account_rep_object'size + 31)/32,
          tdo  => account_TDO);
      account_rep.all := account_rep_object'(
          balance => starting_balance);

      -- 3. Remove rep rights for the exported and
      --    master AD:
      --
      account_untyped := Access_Mgt.Remove(
          AD     => account_rep_untyped,
          rights => Object_Mgt.read_write_rights);

      -- 4. Start a local transaction if there is not
      --    a transaction on the stack:
      --
      if Transaction_Mgt.Get_default_transaction =
          null then
        Transaction_Mgt.Start_transaction;
        trans := true;
      end if;
      begin
```

```
      -- 5. Store the master AD:
      --
      Directory_Mgt.Store(
          name    => master,
          object  => account_untyped,
          aut     => authority);

      -- 6. Passivate the account object itself:
      --
      Passive_Store_Mgt.Update(account_rep_untyped);

      -- 7. Commit any local transaction:
      --
      if trans then
        Transaction_Mgt.Commit_transaction;
      end if;
    exception
      -- 8. If any exception occurs, abort any local
      --    transaction, deallocate the account,
      --    and reraise the exception:
      --
      when others =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        end if;
        Object_Mgt.Deallocate(account_untyped);
        RAISE;

    end;
    RETURN account;

  end if;
end Create_stored_account;


function Get_balance(
    account:  account_AD)
  return Long_Integer_Defs.long_integer
  --
  -- Logic:
  --   1. Amplify rep rights on the account AD.
  --
  --   2. Loop (in case of retry due to a transaction
  --      timestamp conflict).
  --
  --   3. If there is no default transaction,
  --      start a local transaction and flag that
  --      it is started.
  --
  --   4. Reserve the account object to read-lock
  --      the passive version and ensure a clean
  --      and *current* active version.
  --
  --   5. Commit any local transaction, releasing
  --      the lock.
  --
  --   6. Return the balance from the certainly
  --      clean active version.
  --
  --   7. If there is a transaction timestamp
  --      conflict, and if a local transaction was
  --      started, then abort that transaction, loop
  --      back, start a fresh transaction, and try
  --      again.
  --
  --   8. If there is any other exception, then
  --      abort any local transaction and reraise
  --      the exception.
is
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;

  trans:  boolean := false;
    -- True if a local transaction is started.
begin
```

```
  account_untyped := Access_Mgt.Amplify(
      AD     => account_untyped,
      rights => Object_Mgt.read_write_rights,
      tdo    => account_TDO);

  loop
    if Transaction_Mgt.Get_default_transaction =
        null then
      Transaction_Mgt.Start_transaction;
      trans := true;
    end if;
    begin
      Passive_Store_Mgt.Reserve(
          obj  => account_untyped,
          read => true);
      if trans then
        Transaction_Mgt.Commit_transaction;
      end if;
      RETURN account_rep.balance;

    exception
      when System_Exceptions.
           transaction_timestamp_conflict =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        else
          RAISE;

        end if;
      when others =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        end if;
        RAISE;

    end;
  end loop;
end Get_balance;


function Change_balance(
    account:  account_AD;
    amount:   Long_Integer_Defs.long_integer)
  return Long_Integer_Defs.long_integer
  --
  -- Logic:
  --   1. Import the account AD, checking for
  --      change rights and adding rep rights.
  --
  --   2. Loop (in case of retry due to a transaction
  --      timestamp conflict).
  --
  --   3. If there is no default transaction, then
  --      start a local transaction and flag that it
  --      is started.
  --
  --   4. Reserve the account object to write-lock
  --      the passive version and ensure a clean
  --      and *current* active version.
  --
  --   5. If the new balance would be negative, abort
  --      the transaction and exit with an exception.
  --
  --   6. Otherwise, change the balance, update the
  --      passive version, and commit any local
  --      transaction, releasing the lock.
  --
  --   7. If there is a transaction timestamp conflict,
  --      and if a local transaction was started, then
  --      abort that transaction, loop back, start a
  --      fresh transaction, and try again.
  --
  --   8. If there is any other exception, then
  --      abort any local transaction and reraise
  --      the exception.
  --
  -- Notes:
```

```
  --   It might appear that instead of reserving the
  --   object, the implementation could simply compute
  --   the new balance, do the update, and reset the
  --   active version and retry in the infrequent
  --   case that "outdated_object_version" in
  --   "Passive_Store_Mgt" is raised.  However, such
  --   an implementation would base the checking for
  --   an insufficient balance on a possibly obsolete
  --   value, which is unacceptable.
is
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;

  trans:  boolean := false;
    -- True if a local transaction is started.
begin
  account_untyped := Access_Mgt.Import(
          AD     => account_untyped,
          rights => change_rights,
          tdo    => account_TDO);

  loop
    if Transaction_Mgt.Get_default_transaction =
        null then
      Transaction_Mgt.Start_transaction;
      trans := true;
    end if;
    begin
      Passive_Store_Mgt.Reserve(account_untyped);
      if account_rep.balance + amount < zero then
        RAISE insufficient_balance;

      else
        account_rep.balance :=
            account_rep.balance + amount;
        Passive_Store_Mgt.Update(account_untyped);
        if trans then
          Transaction_Mgt.Commit_transaction;
        end if;
        RETURN account_rep.balance;

      end if;
    exception
      when System_Exceptions.
           transaction_timestamp_conflict =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        else
          RAISE;

        end if;
      when others =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        end if;
        RAISE;
    end;
  end loop;
end Change_balance;


procedure Transfer(
    source_account:  account_AD;
    dest_account:    account_AD;
    amount:  Long_Integer_Defs.long_integer)
  --
  -- Logic:
  --   1. Import the account ADs, checking for
  --      change rights and adding rep rights.
  --
  --   2. Loop (in case of retry due to a transaction
  --      timestamp conflict).
  --
  --   3. If there is no default transaction, then
  --      start a local transaction and flag that it
```

```
  --        is started.
  --
  --     4. Reserve the account objects to write-lock
  --        the passive versions and ensure a clean
  --        and *current* active version.
  --
  --     5. If either new balance would be negative, abort
  --        the transaction and exit with an exception.
  --
  --     6. Otherwise, change the balances, update the
  --        passive versions, and commit any local
  --        transaction, releasing the lock.
  --
  --     7. If there is a transaction timestamp conflict,
  --        and if a local transaction was started, then
  --        abort that transaction, loop back, start a
  --        fresh transaction, and try again.
  --
  --     8. If there is any other exception, then
  --        abort any local transaction and reraise
  --        the exception.
is
  source_rep:  account_rep_AD;
  FOR source_rep USE AT source_account'address;
  source_untyped:  System.untyped_word;
  FOR source_untyped USE AT source_account'address;

  dest_rep:  account_rep_AD;
  FOR dest_rep USE AT dest_account'address;
  dest_untyped:  System.untyped_word;
  FOR dest_untyped USE at dest_account'address;

  trans:  boolean := false;
    -- True if a local transaction is started.
begin

  source_untyped := Access_Mgt.Import(
      AD     => source_untyped,
      rights => change_rights,
      tdo    => account_TDO);
  dest_untyped := Access_Mgt.Import(
      AD     => dest_untyped,
      rights => change_rights,
      tdo    => account_TDO);

  loop
    if Transaction_Mgt.Get_default_transaction =
        null then
      Transaction_Mgt.Start_transaction;
      trans := true;
    end if;
    begin
      Passive_Store_Mgt.Reserve(source_untyped);
      Passive_Store_Mgt.Reserve(dest_untyped);
      if source_rep.balance - amount < zero
         or else
         dest_rep.balance + amount < zero
         then
        RAISE insufficient_balance;

      else
        source_rep.balance :=
            source_rep.balance - amount;
        dest_rep.balance   :=
            dest_rep.balance + amount;
        Passive_Store_Mgt.Update(source_untyped);
        Passive_Store_Mgt.Update(dest_untyped);
        if trans then
          Transaction_Mgt.Commit_transaction;
        end if;
        RETURN;

      end if;
    exception
      when System_Exceptions.
           transaction_timestamp_conflict =>
        if trans then
```

```
          Transaction_Mgt.Abort_transaction;
        else
           RAISE;

        end if;
      when others =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        end if;
        RAISE;

    end;
  end loop;
end Transfer;


procedure Destroy_account(
    account:  account_AD)
  --
  -- Logic:
  --   1. Import the account AD, checking for
  --      destroy rights and amplifying rep rights.
  --
  --   2. Loop in case of retry due to timestamp
  --      conflict.
  --
  --   3. If there is no default transaction, then
  --      start a local transaction and flag that it
  --      is started.
  --
  --   4. Reserve the account object to write-lock
  --      the passive version and ensure a clean
  --      and current active version.
  --
  --   5. Check that the account's balance is zero.
  --      If it isn't, raise an exception.  The
  --      block's exception handler will abort
  --      any local transaction.
  --
  --   6. Destroy the account's passive version.
  --
  --   7. Get the name of the account's master
  --      directory entry (if any).  Delete that
  --      directory entry.  Note that other
  --      entries and even a master AD may remain
  --      for the account.
  --
  --   8. If there is a transaction timestamp
  --      conflict, and if a local transaction
  --      was started, then abort that transaction,
  --      loop back, start a fresh transaction,
  --      and try again.
  --
  --   9. If any other exception occurs, abort
  --      any local transaction and reraise the
  --      exception.
  --
  --  10. Deallocate the account's active version.
is
  account_rep:  account_rep_AD;
  FOR account_rep USE AT account'address;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;

  trans:  boolean := false;
     -- True if a local transaction is started.
begin
  account_untyped := Access_Mgt.Import(
      AD     => account_untyped,
      rights => destroy_rights,
      tdo    => account_TDO);
  loop
    if Transaction_Mgt.Get_default_transaction =
        null then
      Transaction_Mgt.Start_transaction;
      trans := true;
    end if;
```

```
      declare
        path_length:  integer := 60;
          -- Initial text length for name assigned
          -- by "Directory_Mgt.Get_name".  If
          -- insufficient, then the value is
          -- increased and the operation is
          -- repeated.
      begin
        Passive_Store_Mgt.Reserve(account_untyped);
        if account_rep.balance /=
            Long_Integer_Defs.zero then
          RAISE balance_not_zero;

        end if;
        Passive_Store_Mgt.Destroy(account_untyped);

        loop
          declare
            path_text:  System_Defs.text(path_length);
          begin
            Directory_Mgt.Get_name(
                obj  => account_untyped,
                name => path_text);  -- out.
            if path_text.length >
               path_text.max_length then
              -- Text was lost.  Retry:
              path_length := path_text.length;
            else
              Directory_Mgt.Delete(path_text);
              EXIT;

            end if;
          exception
            when Directory_Mgt.no_name =>
              EXIT;

          end;
        end loop;
      exception
        when System_Exceptions.
             transaction_timestamp_conflict =>
          if trans then
            Abort_transaction;
          else
            RAISE;

          end if;

        when others =>
          if trans then
            Abort_transaction;
          end if;
          RAISE;

      end;
    EXIT;
    end loop;
    Object_Mgt.Deallocate(account_untyped);
  end Destroy_account;


end Account_Mgt_Ex;
```

X-A.7.10 `Stored_Account_TDO_Init_Ex` Procedure

Initialization procedure for stored account type managers.

```
with Account_Type_Name_Ex,    -- Example package.
     Attribute_Mgt,
     Authority_List_Mgt,
     Directory_Mgt,
     Identification_Mgt,
     Object_Mgt,
     Passive_Store_Mgt,
     Refuse_reset_active_version_Ex, -- Example package
     System,
     System_Defs,
     System_Exceptions,
     Text_Mgt,
     Transaction_Mgt,
     Type_Name_Attribute_Ex,  -- Example package.
     User_Mgt,
     Unchecked_conversion;

procedure Stored_Account_TDO_Init_Ex
  --
  -- Logic:
  --   Initialize TDO for accounts and place it in
  --   the passive store for use by instances of
  --   "Stored_Account_Mgt_Ex" at different nodes.
  --
  --   The account TDO has the OS passive store
  --   attribute and the (example) type name attribute.
  --
  --   Resetting an account's active version or
  --   copying accounts are not allowed outside the
  --   type manager.  Other passive store requests
  --   are allowed.
  -- History:
  --   ??-??-????:  Martin Buchanan, Initial version.
  --   12-01-1987:  Tobias Haas, Removed 'Refuse_reset_active_version'
  --                procedure and placed in separate package.
  --   04-20-1988:  Tobias Haas, Added extractor comments, bstex*.ex
  --   05-06-1988:  Tobias Haas, Modified extractor comments, bstex*.ex
  --   05-20-1988:  Tobias Haas, Added handler for Directory_Mgt.
  --                entry_exists
is
  use Transaction_Mgt;
    -- Import transaction operators.

  account_name:  constant string :=
      "account";
    -- pathname of account tdo.

  account_text:  System_Defs.text(account_name'length);
    -- Pathname is placed in this text before calling
    -- "Directory_Mgt.Store".

  account_TDO:  Object_Mgt.TDO_AD;
    -- TDO for accounts.

  passive_store_impl:
      Passive_Store_Mgt.PSM_attributes_AD;
    -- Implementation of passive store attribute
    -- for accounts.

  type_name_impl:  System.untyped_word;
    -- Implementation of type name attribute
    -- for accounts.

  owner_only:  User_Mgt.protection_set(1);
    -- Protection set that includes only one ID, namely
    -- the type manager's owner.

  authority:  Authority_List_Mgt.authority_list_AD;
    -- Authority list that contains only one ID, namely
    -- the type manager's owner.

  trans:  boolean := false;
    -- Set if local transaction is started.
```

```
  function Untyped_from_PSM_attributes is
      new Unchecked_conversion(
          source => Passive_Store_Mgt.PSM_attributes_AD,
          target => System.untyped_word);


  function Untyped_from_TDO is
      new Unchecked_conversion(
          source => Object_Mgt.TDO_AD,
          target => System.untyped_word);


begin
  Text_Mgt.Set(account_text, account_name);

  account_TDO := Object_Mgt.Create_TDO;

  passive_store_impl := new
      Passive_Store_Mgt.PSM_attributes_object;

  passive_store_impl.reset :=
      Refuse_reset_active_version_Ex.
          Refuse_reset_active_version'subprogram_value;

  passive_store_impl.copy_permitted := false;

  Attribute_Mgt.Store_attribute_for_type(
      tdo       => account_TDO,
      attr_ID   => Passive_Store_Mgt.PSM_attributes_ID,
      attr_impl => Untyped_from_PSM_attributes(
                       passive_store_impl));
  type_name_impl := Account_Type_Name_Ex'package_value;

  Attribute_Mgt.Store_attribute_for_type(
      tdo       => account_TDO,
      attr_ID   => Type_Name_Attribute_Ex.
                   Get_type_name_attr_ID,
      attr_impl => type_name_impl);

   owner_only.length := 1;
   owner_only.entries(1).rights := User_Mgt.access_rights'(
       true, true, true);
   owner_only.entries(1).id := Identification_Mgt.Get_user_id;

   authority := Authority_List_Mgt.Create_authority(owner_only);

  if Transaction_Mgt.Get_default_transaction =
      null then
    Transaction_Mgt.Start_transaction;
    trans := true;
  end if;

  begin
    Directory_Mgt.Store(
        name   => account_text,
        object => Untyped_from_TDO(account_TDO),
        aut    => authority);
    Passive_Store_Mgt.Request_update(
        Untyped_from_TDO(account_TDO));
    Passive_Store_Mgt.Request_update(
        Untyped_from_PSM_attributes(
            passive_store_impl));
    Passive_Store_Mgt.Request_update(
        type_name_impl);

    if trans then
      Transaction_Mgt.Commit_transaction;
    end if;
  exception
    when Directory_Mgt.entry_exists =>
      if trans then
        Transaction_Mgt.Abort_transaction;
      end if;

    when others =>
```

```
      if trans then
        Transaction_Mgt.Abort_transaction;
      end if;
      RAISE;

  end;

end Stored_Account_TDO_Init_Ex;
```

X-A.7.11 `Account_Type_Name_Ex` Package Specification

Type name attribute implementation for stored account type managers.

```
with System,
     Type_Name_Attribute_Ex;

package Account_Type_Name_Ex is
  pragma package_value(Type_Name_Attribute_Ex.Ops);
  --
  -- Function:
  --   Defines the type name attribute for accounts.
  --
  --   A type that supports this attribute has a
  --   printable name.  For example, a directory
  --   listing utility could use this attribute to
  --   print the types of the objects in a
  --   directory.


  function Type_name(
      obj:  System.untyped_word)
    return string;
      -- Name of the "account" object type.
    --
    -- Function:
    --   Returns the type name for account objects.


  pragma external;

end Account_Type_Name_Ex;
```

X-A.7.12 Account_Type_Name_Ex Package Body

Type name attribute implementation for stored account type managers.

```
with System;

package body Account_Type_Name_Ex is


  function Type_name(
      obj:  System.untyped_word)
    return string
  is
  begin
    return "account";
  end Type_name;


end Account_Type_Name_Ex;
```

X-A.7.13 `Type_Name_Attr_Ex` Package Specification

Type name attribute package type.

```
with Attribute_Mgt,
     System;

package Type_Name_Attribute_Ex is
  --
  -- Function:
  --   Define an attribute that returns a type's name.
  --
  --   A type that supports the *type name* attribute has a
  --   printable name.  For example, a directory listing utility
  --   could use the attribute to print the types of the objects
  --   in a directory.

  function Get_type_name_attr_ID
    return Attribute_Mgt.attribute_ID_AD;
      -- Type name attribute ID, with type rights.
    --
    -- Function:
    --   Returns the type name attribute's attribute ID.



  package Ops is
    pragma package_type("typnamattr");
      --
      -- Function:
      --   Provide "Type_name" attribute call.


    function Type_name(
        obj:  System.untyped_word)
          -- Any object that supports
          -- the type name attribute.
      return string;  -- Name of the object's type.
      pragma interface(value, Type_name);
      --
      -- Function:
      --   Returns a printable name for an object's type.


  end Ops;



  pragma external;

end Type_Name_Attribute_Ex;
```

X-A.7.14 Type_Name_Attr_Ex Package Body

Type name attribute package type.

```
with Attribute_Mgt,
     System_Defs;

package body Type_Name_Attribute_Ex is


  type_name_attr_ID:  constant
      Attribute_Mgt.attribute_ID_AD := null;
    pragma bind(type_name_attr_ID,
                "typnamattr");
    -- Attribute ID is retrieved at link time using the
    -- specified pathname.  Should have store rights.


  function Get_type_name_attr_ID
    return Attribute_Mgt.attribute_ID_AD
  is
  begin
    return type_name_attr_ID;
  end Get_type_name_attr_ID;


  package body Ops is
    --
    -- Logic:
    --   Attribute packages have null bodies.


  end Ops;


end Type_Name_Attribute_Ex;
```

X-A.7.15 `Type_Name_Attribute_Init_Ex` Procedure

Creates the type name attribute ID.

```
with
  Attribute_Mgt,
  Conversion_Support_Ex,
  Directory_Mgt,
  Passive_Store_Mgt,
  System_Defs,
  Transaction_Mgt;

 procedure Type_Name_Attribute_Init_Ex is
   --
   -- Function:
   --   o Create new attribute.
   --
   --   o Store new attribute. If attribute already
   --     exists, all changes are rolled back and the
   --     procedure exists
   --
   --   o Update new attribute.
   --
   -- History:
   --   05-10-1988:   Tobias Haas:   Initial version.

   typ_nam_attr_ID_AD: Attribute_Mgt.attribute_ID_AD;
      -- New attribute.

 begin
   Transaction_Mgt.Start_transaction;
      -- Transaction ensures that both operations, Store and
      -- Update, will take place together or not at all.
   begin
     typ_nam_attr_ID_AD := Attribute_Mgt.Create_attribute_ID(
         type_specific => true);
        -- Create new attribute.

     Directory_Mgt.store(
         name   => System_Defs.text'(10, 10,"typnamattr"),
         object => Conversion_Support_Ex.Untyped_from_attribute_ID(
                   typ_nam_attr_ID_AD));
       -- Store attribute. If attribute already exists, this
       -- operation will cause the Directory_Mgt.entry_exists
       -- exception to be raised.

     Passive_Store_Mgt.Request_update(Conversion_Support_Ex.
                              Untyped_from_attribute_ID(typ_nam_attr_ID_AD));
     Transaction_Mgt.Commit_transaction;
       -- Commit transaction after successful completion of
       -- both operations.

   exception
     when Directory_Mgt.entry_exists =>
       Transaction_Mgt.Abort_transaction;
         -- If entry exits, roll back any changes.

     when others =>
       Transaction_Mgt.Abort_transaction;
       RAISE;

   end;
 end Type_Name_Attribute_Init_Ex;
```

X-A.7.16 Refuse_Reset_Active_Version_Ex Package Specification

Type-specific implementation for stored accounts.

```
with System,
     System_Exceptions,
     Passive_Store_Mgt;

package Refuse_reset_active_version_Ex is

  procedure Refuse_reset_active_version(
      obj:  System.untyped_word);
    --
    -- Function:
    --   Handles requests to reset an account's active
    --   version by refusing such requests.
    --
    pragma external;

    pragma subprogram_value(
        Passive_Store_Mgt.
            Type_specific_reset_active_version,
        Refuse_reset_active_version);

end Refuse_reset_active_version_Ex;
```

X-A.7.17 `Refuse_Reset_Active_Version_Ex` Package Body

Type-specific implementation for stored accounts.

```
with System,
     System_Exceptions,
     Passive_Store_Mgt;

package body Refuse_reset_active_version_Ex is

-- History:
--   12-01-87:  Tobias Haas, initial version.
--   04-20-87:  Tobias Haas, added extractor comments bstex*.ex

  procedure Refuse_reset_active_version(
      obj:  System.untyped_word)
  is
    --
    -- Function:
    --   Handles requests to reset an account's active
    --   version by refusing such requests.
    --

  begin

    RAISE System_Exceptions.operation_not_supported;

  end Refuse_reset_active_version;

end Refuse_reset_active_version_Ex;
```

X-A.7.18 Account_Mgt_Ex (Distributed) Package Body

Package body of the distributed account manager.

```
with
  Access_Mgt,
  Attribute_Mgt,
  Authority_List_Mgt,
  Directory_Mgt,
  Distr_Acct_Call_Stub_Ex,
  Long_Integer_Defs,
  Object_Mgt,
  Passive_Store_Mgt,
  Semaphore_Mgt,
  System,
  System_Defs,
  System_Exceptions,
  Transaction_Mgt;

 package body Account_Mgt_Ex is
   --
   -- Logic:
   --   This is an implementation of the distributed
   --   account manager. It follows the single activation
   --   model. It has the following characteristics:
   --
   --     * All operations on accounts are centralized in
   --       one home job. The home job is created at the node
   --       where the first call to this package is made.
   --
   --     * Accounts can be stored anywhere on the system.
   --
   --     * Initialization, (creating the TDO, the server,
   --       the service, installing the server, and setting up
   --       the homomorph template) happen when the package is
   --       eleborated.
   --
   --     * All synchronization is centralized in the
   --       home job: Transactions are used to synchronize accross
   --       job boundaries and semaphores to synchronize between
   --       different processes  inside one job.
   --
   --     * This code is used in the home job and in all
   --       other jobs. In the home job operations are
   --       done directly. In all other jobs a call stub
   --       package is called that issues RPCs
   --       to the home job to perform the actual operation.
   --
   --     * The following picture
   --       illustrates the structure of the distributed
   --       implementation. Boxes represent independent jobs
   --       that may run on any node. The names in the boxes
   --       are the names of the packages.
   --
   --       +---------------+     +---------------+
   --       |Account_Mgt_Ex|     |Account_Mgt_Ex|
   --       |               |     |               |
   --       |  Distr_Acct_  |     |  Distr_Acct_  |
   --       |   Call_Stub   |     |   Call_Stub   |
   --       +---------------+     +---------------+
   --          Application           Application
   --             Job                   Job
   --
   --
   --               +---------------+
   --               |  Distr_Acct_  |
   --               |  Server_Stub  |
   --               |               |
   --               |Account_Mgt_Ex|
   --               +---------------+
   --                  Server Job
   --                  (Home Job)
   --
   --
   --     * ADs to the TDO  and the account service are created
   --       by an initialization routine called Distr_acct_init
   --       and stored with pathnames. They are retrieved by the
```

```
--       various models at link-time.
--
-- Exceptions:
--   no_server_installed:
--     Server for accounts is not installed.
--
-- History:
--   01-31-88: Tobias Haas, Initial version.
--   06-08-88: Tobias Haas, Design revision.

use Long_Integer_Defs,  -- Import "long_integer", "zero",
                        -- arithmetic, and relational
                        -- operators.
    System,             -- Import ordinal operators.
    Transaction_Mgt;    -- Import transaction calls.


account_TDO: constant Object_Mgt.TDO_AD := null;
  pragma bind(account_TDO, "account");
  -- Constant AD to account TDO. Initially null.
  -- Filled in at link-time.

type account_rep_object is
  -- Representation of an account.
  record
    lock:  Semaphore_Mgt.semaphore_AD;
      -- Locking area
    is_homomorph:  boolean;
      -- If false identifies the object
      -- as the active version; if true
      -- as a homomorph.
    balance: Long_Integer_Defs.long_integer;
      -- Starting balance.
  end record;
    FOR account_rep_object USE
    record AT mod 32;
      lock          at  0  range  0 .. 31;
      is_homomorph  at  4  range  0 ..  7;
      balance       at  8  range  0 .. 63;
    end record;
type account_rep_AD is access account_rep_object;
  pragma access_kind(account_rep_AD, AD);
  -- Private view of an account.



----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                        IS_ACCOUNT                                --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------

function Is_account(
    obj:  System.untyped_word)
  return boolean
  --
  -- Logic:
  --   If "obj" is not null, retrieve the object's
  --   TDO and check whether it is the account's TDO.
  --
is
  use Object_Mgt;  -- Import "=" for type "TDO_AD".
begin
  return obj /= System.null_word
      and then
      Object_Mgt.Retrieve_TDO(obj) = account_TDO;
end Is_account;

----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                        CREATE_ACCOUNT                            --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------
```

```
function Create_account(
    starting_balance:
        Long_Integer_Defs.long_integer :=
        Long_Integer_Defs.zero)
  return account_AD
  --
  -- Logic:
  --   Creates an account by allocating an object
  --   of type account. Storing the account is the
  --   responsibility of the caller. Accounts can
  --   be created in any account.
  --
  --   1. Check initial balance.
  --
  --   2. Allocate and initialize the account
  --      object.
  --
  --   3. Remove rep rights for the exported and
  --      master AD.
  --
  --   4. If any exception occurs, deallocate the object
  --      and return.
  --
is
  account:          account_AD;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;
    -- Account with no rep rights, viewed with
    -- either of two types.

  account_rep:          account_rep_AD;
  account_rep_untyped:  System.untyped_word;
  FOR account_rep_untyped use AT
      account_rep'address;
    -- Account with rep rights, viewed with
    -- either of two types.

  trans:  boolean := false;
    -- True if a local transaction has been
    -- started.
begin
  -- 1. Check initial balance:
  --
  if starting_balance <
      Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    -- 2. Allocate and initialize the
    --    account object:
    --
    account_rep_untyped := Object_Mgt.Allocate(
        size => (account_rep_object'size+31)/32,
        tdo  => account_TDO);

    begin
      account_rep.all := account_rep_object'(
          lock         => null,
          is_homomorph => false,
          balance      => starting_balance);
      -- 3. Remove rep rights for the exported and
      --    master AD:
      --
      account_untyped := Access_Mgt.Remove(
          AD     => account_rep_untyped,
          rights => Object_Mgt.read_write_rights);

    exception
      -- 4. If an exception occurs, deallocate the account
      --    and reraise the exception:
      --
      when others =>

        Object_Mgt.Deallocate(account_untyped);
        RAISE;

```

```
      end;
      RETURN account;

    end if;

  end Create_account;

----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                      CREATE_STORED_ACCOUNT                       --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------


  function Create_stored_account(
      starting_balance:
          Long_Integer_Defs.long_integer :=
          Long_Integer_Defs.zero;
      master:  System_Defs.text;
      authority:
          Authority_List_Mgt.authority_list_AD :=
          null)
    return account_AD
    --
    -- Logic:
    --   Any job can create accounts. In order to
    --   ensure that no multiple active versions of
    --   any account exist the active version is
    --   deallocated as soon as it has been
    --   passivated. Passivating  the master AD
    --   and deallocating the active version
    --   are enclosed in a transaction.
    --   These are the steps:
    --
    --   1. Check initial balance.
    --
    --   2. Allocate and initialize the account
    --      object.
    --
    --   3. Remove rep rights for the exported and
    --      master AD.
    --
    --   4. Start a local transaction if there is
    --      not a transaction on the stack.
    --
    --   5. Create a master AD. Use "Store". This also
    --      sets the object's authority list.
    --
    --   6. Passivate the account.
    --
    --   7. Deallocate the active version of the
    --      account.
    --
    --   8. Commit any local transaction.
    --
    --   9. If an exception occurs, abort any local
    --      transaction, deallocate the account
    --      and reraise the exception.
    --
  is
    account:          account_AD;
    account_untyped:  System.untyped_word;
    FOR account_untyped USE AT account'address;
      -- Account with no rep rights, viewed with
      -- either of two types.


    account_rep:          account_rep_AD;
    account_rep_untyped:  System.untyped_word;
    FOR account_rep_untyped use AT
        account_rep'address;
      -- Account with rep rights, viewed with
      -- either of two types.
```

```
  trans:  boolean := false;
    -- True if a local transaction has been
    -- started.
begin
  -- 1. Check initial balance:
  --
  if starting_balance <
      Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    -- 2. Allocate and initialize the
    --    account object:
    --
    account_rep_untyped := Object_Mgt.Allocate(
        size => (account_rep_object'size+31)/32,
        tdo  => account_TDO);
    account_rep.all := account_rep_object'(
        lock         => null,
          -- Null because ''lock'' is not present
          -- in passive version.
        is_homomorph => false,
        balance      => starting_balance);


    -- 3. Remove rep rights for the exported and
    --    master AD:
    --
    account_untyped := Access_Mgt.Remove(
        AD     => account_rep_untyped,
        rights => Object_Mgt.read_write_rights);


    -- 4. Start a local transaction if there is
    --    not one on the stack:
    --
    if Transaction_Mgt.Get_default_transaction =
        null then
      Transaction_Mgt.Start_transaction;
      trans := true;
    end if;


    begin
      -- This block controls the scope of
      -- the exception handler.

      -- 5. Create the master AD:
      --
      Directory_Mgt.Store(
          name   => master,
          object => account_untyped,
          aut    => authority);

      -- 6. Passivate the representation of the account:
      --
      Passive_Store_Mgt.Update(account_rep_untyped);

      -- 7. Deallocate the active version of the
      --    account:
      --
      Object_Mgt.Deallocate(account_rep_untyped);

      -- 8. Commit any local transaction.
      --
      if trans then
        Transaction_Mgt.Commit_transaction;
      end if;

    exception

      -- 9. If an exception occurs, abort any local
      --    transaction, deallocate the account and
      --    reraise the exception:
      --
      when others =>
        if trans then
```

```
            Transaction_Mgt.Abort_transaction;
          end if;
          Object_Mgt.Deallocate(account_rep_untyped);
          RAISE;

      end;
      RETURN account;

    end if;
  end Create_stored_account;


----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                      GET_BALANCE                                 --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------

  function Get_balance(
      account:  account_AD)
    return Long_Integer_Defs.long_integer
    --
    -- Logic:
    --   1. Amplify rep rights on the account AD.
    --
    --   2. If "is_homomorph" is true:
    --
    --        * Call the call stub.
    --
    --   3. If "is_homomorph" is false:
    --
    --        * Start transaction if there is not
    --          one on the stack.
    --
    --        * Lock account with a semaphore.
    --          (Deadlock is avoided by the
    --          transaction timeout.)
    --
    --        * Read current balance.
    --
    --        * If an exeception occurs release the
    --          account and abort any local transaction.
    --
    --        * Release the object and commit any local
    --          transaction.

  is
    account_rep:      account_rep_AD;
      -- Account with rep rights.

    account_rep_untyped:  System.untyped_word;
      FOR account_rep_untyped USE AT account_rep'address;
      -- untyped view of account with rep rights.

    account_no_rep_untyped:  System.untyped_word;
      FOR account_no_rep_untyped USE AT account'address;
      -- Untyped view of account with no rep rights.

    current_balance:  Long_Integer_Defs.long_integer;
      -- Current balance.

    trans:  boolean := false;
      -- Is true if there is a local transaction.

  begin
    account_rep_untyped := account_no_rep_untyped;

    -- 1. Amplify rep rights:
    --
    account_rep_untyped := Access_Mgt.Amplify(
        AD     => account_rep_untyped,
        rights => Object_Mgt.read_write_rights,
        tdo    => account_tdo);

    if account_rep.is_homomorph then
```

```
      -- 2. We have a homomorph:
      --

      -- Call the call stub:
      --
      RETURN Distr_Acct_Call_Stub_Ex.
            Get_balance(account);

    else

      -- 3. We are in the home job for accounts:
      --

      -- Start a local transaction if there is not one
      -- on the stack:
      --
      if Transaction_Mgt.Get_default_transaction = null
           then
        Transaction_Mgt.Start_transaction;
        trans := true;
      end if;

      begin
        -- "P" locks the account object. If another
        -- process has already locked the object wait
        -- until the object is released. Transaction
        -- timeout prevents a deadlock. (A finite timeout
        -- value has to be set at node initialization.)
        --
        Semaphore_Mgt.P(
            semaphore => account_rep.lock);

        begin
          -- Read current balance:
          --
          current_balance := account_rep.balance;

          -- Release the account:
          --
          Semaphore_Mgt.V(
              semaphore => account_rep.lock);

          -- Commit any local transaction:
          --
          if trans then
            Transaction_Mgt.Commit_transaction;
          end if;

          RETURN current_balance;

        exception
          -- Release the object:
          --
          when others =>
            Semaphore_Mgt.V(semaphore =>
                account_rep.lock);
          RAISE;

        end;

      exception
        -- Abort any local transaction:
        --
        when others =>
          if trans then
            Transaction_Mgt.Abort_transaction;
          end if;
          RAISE;
      end;

    end if;

  end Get_balance;

----------------------------------------------------------------------
----------------------------------------------------------------------
```

```
--                                                                    --
--                          CHANGE_BALANCE                            --
--                                                                    --
------------------------------------------------------------------------
------------------------------------------------------------------------


  function Change_balance(
      account:  account_AD;
      amount:   Long_Integer_Defs.long_integer)
    return Long_Integer_Defs.long_integer
    --
    -- Logic:
    --   1. Check "account" for change rights and add rep
    --      rights.
    --
    --   2. If "is_homomorph" is true make a remote call.
    --
    --   3. If "is_homomorph" is false update the balance
    --      and return the new balance.
    --
  is
    account_rep:      account_rep_AD;
      -- Account with rep rights.

    account_rep_untyped:  System.untyped_word;
      FOR account_rep_untyped USE AT account_rep'address;
      -- untyped view of account with rep rights.

    account_no_rep_untyped:  System.untyped_word;
      FOR account_no_rep_untyped USE AT account'address;
      -- Untyped view of account with no rep rights.

    current_balance:  Long_Integer_Defs.long_integer;
      -- Current balance.

    trans:  boolean := false;
      -- Is true if there is a local transaction.

  begin
    account_rep_untyped := account_no_rep_untyped;
    account_rep_untyped := Access_Mgt.Import(
        AD     => account_rep_untyped,
        rights =>   change_rights,
        tdo    => account_TDO);

    if account_rep.is_homomorph then
      RETURN Distr_Acct_Call_Stub_Ex.Change_balance(
          account => account,
          amount  => amount);

    else
      if Transaction_Mgt.Get_default_transaction = null
          then
        Transaction_Mgt.Start_transaction;
        trans := true;
      end if;

      begin
        Semaphore_Mgt.P(account_rep.lock);

        begin
          if account_rep.balance + amount < zero then
            RAISE insufficient_balance;

          else
            account_rep.balance := account_rep.balance +
                amount;
            Passive_Store_Mgt.Update(account_rep_untyped);
            Semaphore_Mgt.V(account_rep.lock);

            if trans then
              Transaction_Mgt.Commit_transaction;
            end if;
            RETURN account_rep.balance;

          end if;
```

```

        exception
          when others =>
            Semaphore_Mgt.V(semaphore =>
                account_rep.lock);
            RAISE;

        end;

      exception

        when others =>
          if trans then
            Transaction_Mgt.Abort_transaction;
          end if;
          RAISE;

      end;

    end if;

  end Change_balance;

----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                        TRANSFER                                  --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------

  procedure Transfer(
      source_account:  account_AD;
      dest_account:    account_AD;
      amount:          Long_Integer_Defs.long_integer)
    --
    -- Logic:
    --   1. Check rights on both ADs and add rep rights.
    --
    --   2. If "is_homomorph" is true make a remote call.
    --      If "is_homomorph" is false proceed with the
    --      transfer.
    --
    --   3. If any of the resultant balances are negative
    --      raise "insufficient_balance".
    --
  is
    source_rep:  account_rep_AD;

    source_rep_untyped:  System.untyped_word;
      FOR source_rep_untyped USE AT source_rep'address;

    source_no_rep_untyped:  System.untyped_word;
      FOR source_no_rep_untyped USE AT source_account'address;

    dest_rep:  account_rep_AD;

    dest_rep_untyped:  System.untyped_word;
      FOR dest_rep_untyped USE AT dest_rep'address;

    dest_no_rep_untyped:  System.untyped_word;
      FOR dest_no_rep_untyped USE AT dest_account'address;

    trans:  boolean := false;

  begin
    source_rep_untyped := source_no_rep_untyped;
    source_rep_untyped := Access_Mgt.Import(
        AD     => source_rep_untyped,
        rights => change_rights,
        tdo    => account_TDO);

    dest_rep_untyped := dest_no_rep_untyped;
    dest_rep_untyped := Access_Mgt.Import(
        AD     => dest_rep_untyped,
        rights => change_rights,
        tdo    => account_TDO);
```

```
    if source_rep.is_homomorph then
      -- Only one of the accounts has to be checked.
      Distr_Acct_Call_Stub_Ex.Transfer(
          source_account => source_account,
          dest_account   => dest_account,
          amount         => amount);
      RETURN;

    else
      if Transaction_Mgt.Get_default_transaction =
          null then
        Transaction_Mgt.Start_transaction;
      end if;

      begin
        Semaphore_Mgt.P(
            semaphore => source_rep.lock);

        begin
          Semaphore_Mgt.P(
              semaphore => dest_rep.lock);

          begin
            if (source_rep.balance - amount < zero)
                or (dest_rep.balance - amount < zero)
                then
              RAISE insufficient_balance;

            else
              source_rep.balance :=
                  source_rep.balance - amount;
              dest_rep.balance :=
                  dest_rep.balance - amount;
              Passive_Store_Mgt.Update(source_rep_untyped);
              Passive_Store_Mgt.Update(dest_rep_untyped);
              if trans then
                Transaction_Mgt.Commit_transaction;
              end if;

            end if;
            RETURN;

          exception
            when others =>
              Semaphore_Mgt.V(
                  semaphore => dest_rep.lock);
            RAISE;

          end;
        exception
          when others =>
            Semaphore_Mgt.V(
                semaphore => source_rep.lock);
          RAISE;

        end;
      exception
        when others =>
          if trans then
            Transaction_Mgt.Abort_transaction;
          end if;
        RAISE;

      end;

    end if;

  end Transfer;

-----------------------------------------------------------------------
-----------------------------------------------------------------------
--                                                                   --
--                          DESTROY_ACCOUNT                          --
--                                                                   --
-----------------------------------------------------------------------
-----------------------------------------------------------------------
```

```

procedure Destroy_account(
    account:  account_AD)
    --
    -- Logic:
    --   1. Check rights on "account". Add rep rights.
    --
    --   2. If "is_homomorph" is true make a remote call.
    --
    --   3. If "is_homomorph" is false proceed.
    --
    --   4. Start a local transaction if there is not one
    --      on the stack.
    --
    --   5. lock the object with a semaphore
    --
    --   6. Check that the account balance is zero,
    --      otherwise raise an exception.
    --
    --   7. Destroy the account's passive version.
    --
    --   8. Get the name of the object's master directory
    --      entry. (if any) Remove that entry. Note that
    --      other entries and even a master AD may remain.
    --
    --   9. If any exception occurs abort any local
    --      transaction and reraise the exception.
    --
    --  10. Deallocate the account's active version.
    --
  is
    account_rep:  account_rep_AD;

    account_rep_untyped:  System.untyped_word;
      FOR account_rep_untyped USE AT account_rep'address;

    account_no_rep_untyped:  System.untyped_word;
      FOR account_no_rep_untyped USE AT account'address;

    trans:  boolean := false;

  begin
    account_rep_untyped := account_no_rep_untyped;

    account_rep_untyped := Access_Mgt.Import(
        AD     => account_rep_untyped,
        rights => destroy_rights,
        tdo    => account_TDO);

    if account_rep.is_homomorph then
      Distr_Acct_Call_Stub_Ex.Destroy_account(
          account => account);
      RETURN;

    else
      if Transaction_Mgt.Get_default_transaction =
           null then
        Transaction_Mgt.Start_transaction;
        trans := true;
      end if;

      begin
        Semaphore_Mgt.P(
            semaphore => account_rep.lock);

        declare
          path_length:  integer := 60;

        begin
          if account_rep.balance /=
              Long_Integer_Defs.zero then
            RAISE balance_not_zero;

          end if;
          Passive_Store_Mgt.Destroy(account_rep_untyped);

          loop
```

```
            declare
              path_text:  System_Defs.text(path_length);

            begin
              Directory_Mgt.Get_name(
                  obj  => account_rep_untyped,
                  name => path_text);

              if path_text.length >
                  path_text.max_length then
                -- text was lost. Try again.
                path_length := path_text.length;

              else
                Directory_Mgt.Delete(path_text);
                EXIT;

              end if;

            exception
              when Directory_Mgt.no_name =>
                EXIT;

            end;

          end loop;
          Semaphore_Mgt.Destroy_semaphore(
              semaphore => account_rep.lock);
          Object_Mgt.Deallocate(account_rep_untyped);

        exception
          when others =>
            Semaphore_Mgt.V(
                semaphore => account_rep.lock);
            RAISE;

        end;

      exception
        when others =>
          if trans then
            Transaction_Mgt.Abort_transaction;
          end if;
          RAISE;

      end;

    end if;

  end Destroy_account;

end Account_Mgt_Ex;
```

X-A.7.19 Distr_Acct_Call_Stub_Ex Package Specification

Call stub for the distributed account manager. Routes the type manager's requests.

```
with
  Account_Mgt_Ex,
  Authority_List_Mgt,
  Long_Integer_Defs,
  Object_Mgt,
  System,
  System_Defs;

package Distr_Acct_Call_Stub_Ex is
  --
  -- Function:
  --   Call stub for distributed accounts
  --   type manager. Packs parameters into buffers and
  --   makes RPCs. Unpacks the results buffer
  --   and returns results to front end of type
  --   manager. "Is_account", "Create_account",
  --   "Create_stored_account" are always forwarded
  --   directly to the core and are therefore not
  --   needed in the call stub.
  --
  -- Calls:
  --
  --   Get_balance          - Returns an account's
  --                          balance.
  --
  --   Change_balance       - Changes an account's
  --                          balance.
  --
  --   Transfer             - Moves an amount between
  --                          accounts.
  --
  --   Destroy_account      - Destroys an account.
  --


  function Get_balance(
      account:  Account_Mgt_Ex.account_AD)
    return Long_Integer_Defs.long_integer;
    pragma protected_return(Get_balance);



  function Change_balance(
      account: Account_Mgt_Ex.account_AD;
      amount:   Long_Integer_Defs.long_integer)
    return Long_Integer_Defs.long_integer;
    pragma protected_return(Change_balance);



  procedure Transfer(
      source_account: Account_Mgt_Ex.account_AD;
      dest_account:   Account_Mgt_Ex.account_AD;
      amount:          Long_Integer_Defs.long_integer);
    pragma protected_return(Transfer);



  procedure Destroy_account(
      account: Account_Mgt_Ex.account_AD);
    pragma protected_return(Destroy_account);


  pragma external;
    -- Required if this package is used with the
    -- "virtual" compilation model, which supports
    -- multiple domains and multiple subsystems.

private

  type account_object is
    -- Empty dummy record. The object representation
    -- is defined in the package body.
```

```
      record
        null;
      end record;

    pragma external;

  end Distr_Acct_Call_Stub_Ex;
```

X-A.7.20 `Distr_Acct_Call_Stub_Ex` Package Body

Call stub for the distributed account manager. Routes the type manager's requests.

```
with
  Account_Mgt_Ex,
  Distr_Acct_Server_Stub_Ex,
  Job_Types,
  Long_Integer_Defs,
  Object_Mgt,
  RPC_Call_Support,
  RPC_Mgt,
  Semaphore_Mgt,
  System_Defs;

 package body Distr_Acct_Call_Stub_Ex is

  type account_rep_object is
    -- Representation of an account.
    record
      lock:  Semaphore_Mgt.semaphore_AD;
        -- Locking area
      is_homomorph:  boolean;
        -- If false identifies the object
        -- as the active version; if true
        -- as a homomorph.
      balance: Long_Integer_Defs.long_integer;
        -- Starting balance.
    end record;

      FOR account_rep_object USE
      record AT mod 32;
        lock          at  0  range  0 .. 31;
        is_homomorph  at  4  range  0 ..  7;
        balance       at  8  range  0 .. 63;
      end record;

  type account_rep_AD is access account_rep_object;
    pragma access_kind(account_rep_AD, AD);
    -- Private view of an account.

  service:  constant RPC_Mgt.RPC_service_AD := null;
    -- Distributed account service.
    -- This is a constant but not really null. Will
    -- be filled in with an AD retrieved by the linker.

    pragma bind(service, "account_service");
       -- Bind to account service



----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                         GET_BALANCE                              --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------

  function Get_balance(
      account:  Account_Mgt_Ex.account_AD)
    return Long_Integer_Defs.long_integer
    --
    -- Logic:
    --   Pack Parameters into buffer and make RPC.
    --   "Get_balance" has ordinal value 1
    --
  is
    account_untyped:  System.untyped_word;
      FOR account_untyped USE AT account'address;
      -- untyped view of account

    current_balance:  Long_Integer_Defs.long_integer;
      -- Current balance.

    parameters, results:  Distr_Acct_Server_Stub_Ex.buffer;
      -- Buffers for parameters and results.
```

```

    length:  System.ordinal;
      -- Used in remote call to hold actual length of
      -- results buffer.

  begin
    -- Pack parameter buffer:
    --
    parameters := Distr_Acct_Server_Stub_Ex.buffer'(
        first_word  => account_untyped,
        second_word => System.null_word,
          -- irrelevant
        amount      => Long_Integer_Defs.zero);
          -- irrelevant

    -- Make the RPC:
    --
    length := RPC_Call_Support.Remote_call(
        service        => service,
        target_proc    => 1,
        param_buf      => parameters'address,
        param_length   => parameters'size,
        ADs_present    => true,
        results_buf    => results'address,
        results_length => results'size);
      --  "length" is not used here.

    current_balance := results.amount;
    RETURN current_balance;

  end Get_balance;

----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                        CHANGE_BALANCE                            --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------


  function Change_balance(
      account:  Account_Mgt_Ex.account_AD;
      amount:   Long_Integer_Defs.long_integer)
    return Long_Integer_Defs.long_integer
  is
    account_untyped:  System.untyped_word;
      FOR account_untyped USE AT account'address;
      -- untyped view of account.

    parameters, results:  Distr_Acct_Server_Stub_Ex.buffer;
      -- Buffers for parameters and results.

    length:  System.ordinal;
      -- Used in remote call to hold actual length of
      -- results buffer.

  begin
    parameters := Distr_Acct_Server_Stub_Ex.buffer'(
        first_word  => account_untyped,
        second_word => System.null_word,
          -- irrelevant
        amount      => amount);

    length := RPC_Call_Support.Remote_call(
        service        => service,
        target_proc    => 2,
        param_buf      => parameters'address,
        param_length   => parameters'size,
        ADs_present    => true,
        results_buf    => results'address,
        results_length => results'size);
    RETURN results.amount;

  end Change_balance;

----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
--                                                                  --
--                        TRANSFER                                  --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------

  procedure Transfer(
      source_account:  Account_Mgt_Ex.account_AD;
      dest_account:    Account_Mgt_Ex.account_AD;
      amount:          Long_Integer_Defs.long_integer)
  is
    source_untyped:  System.untyped_word;
      FOR source_untyped USE AT source_account'address;

    dest_untyped:  System.untyped_word;
      FOR dest_untyped USE AT dest_account'address;

    length:  System.ordinal;

    parameters, results:  Distr_Acct_Server_Stub_Ex.buffer;

  begin
    parameters := Distr_Acct_Server_Stub_Ex.buffer'(
        first_word  => source_untyped,
        second_word => dest_untyped,
        amount      => amount);

    length := RPC_Call_Support.Remote_call(
        service        => service,
        target_proc    => 3,
        param_buf      => parameters'address,
        param_length   => parameters'size,
        ADs_present    => true,
        results_buf    => results'address,
        results_length => results'size);
    RETURN;

  end Transfer;

----------------------------------------------------------------------
----------------------------------------------------------------------
--                                                                  --
--                        DESTROY_ACCOUNT                           --
--                                                                  --
----------------------------------------------------------------------
----------------------------------------------------------------------

procedure Destroy_account(
    account:  Account_Mgt_Ex.account_AD)
  is
    account_untyped:  System.untyped_word;
      FOR account_untyped USE AT account'address;

    parameters, results:  Distr_Acct_Server_Stub_Ex.buffer;

    length:  System.ordinal;

  begin
    parameters := Distr_Acct_Server_Stub_Ex.buffer'(
        first_word  => account_untyped,
        second_word => System.null_word,
          -- irrelevant.
        amount      => Long_Integer_Defs.zero);
          -- irrelevant.
    length := RPC_Call_Support.Remote_call(
        service        => service,
        target_proc    => 4,
        param_buf      => parameters'address,
        param_length   => parameters'size,
        ADs_present    => true,
        results_buf    => results'address,
        results_length => results'size);
    RETURN;

  end Destroy_account;
```

```
end Distr_Acct_Call_Stub_Ex;
```

X-A.7.21 `Distr_Acct_Server_Stub_Ex` Package Specification

Server stub for distributed account manager. Receives and forwards RPC's.

```
with
  Long_Integer_Defs,
  System;

package Distr_Acct_Server_Stub_Ex
  --
  -- Function:
  --   This package contains the
  --   server stub procedure for distributed
  --   account services.
  --   Corresponds to RPC_Mgt.server_stub.
  --
is
  type buffer is
    -- Buffer used for remote calls.
    record
      first_word:   System.untyped_word;
      second_word:  System.untyped_word;
      amount:       Long_Integer_Defs.long_integer;
    end record;


  FOR buffer USE
    record AT mod 32;
      first_word   at  0  range  0 .. 31;
      second_word  at  4  range  0 .. 31;
      amount       at  8  range  0 .. 63;
    end record;

  -- Exceptions:
  --   System_Exceptions.operation_not_supported is raised when
  --   a target procedure outsice the range 0 .. 4 is specified.
  --

  procedure server_stub(
    --
    -- Function:
    --  Depending on the value of "target_proc",
    --  upacks the parameter buffer, makes the
    --  corresponding call to "Distr_SA_Account_Mgt_Ex",
    --  packs the results buffer, and returns.
    --
      target_proc:             System.short_ordinal;
        -- The number of the procedure to be called.
        -- Has to be in the range 0 .. 4. The
        -- assignments are as follows:
        --
        --   0:   Calls Passive_Store_Mgt.Set_home_job
        --        in order to initialize the server.
        --
        --   1:   Calls Account_Mgt_Ex.Get_balance.
        --
        --   2:   Calls Account_Mgt_Ex.Change_balance.
        --
        --   3:   Calls Account_Mgt_Ex.Transfer.
        --
        --   4:   Calls Account_Mgt_Ex.Destroy_account.
        --
      version:                 System.ordinal;
        -- Not used.
      param_buf:               System.address;
        -- Address of parameter buffer.
      param_length:            System.ordinal;
        -- length of parameter buffer.
      results_buf:             System.address;
        -- Address of results buffer.
      results_length:  in out System.ordinal;
        -- Length of results buffer.
      ADs_returned:        out boolean);
        -- Are any ADs returned. If false, speeds
        -- up the call.

  pragma external;
```

```

end Distr_Acct_Server_Stub_Ex;
```

X-A.7.22 `Distr_Acct_Server_Stub_Ex` Package Body

Server stub for distributed account manager. Receives and forwards RPC's.

```
with
  Account_Mgt_Ex,
  Long_Integer_Defs,
  Object_Mgt,
  Passive_Store_Mgt,
  System,
  System_Exceptions;

package body Distr_Acct_Server_Stub_Ex is
--
-- Function:
--    This package contains the server stub
--    procedure for the distributed account
--    service.
--
-- History:
--    01-31-88: Tobias Haas, Initial version.
--    04-07-88: Extensive Revision of design.


  procedure server_stub(
      target_proc:              System.short_ordinal;
      version:                  System.ordinal;
      param_buf:                System.address;
      param_length:             System.ordinal;
      results_buf:              System.address;
      results_length:  in out System.ordinal;
      ADs_returned:       out boolean)
    --
    -- Function:
    --    Procedure called by the account server
    --    that unpacks the parameter buffer and
    --    makes the appropriate calls.
    --
    -- Logic:
    --   Depending on "target_proc" unpacks "param_buf"
    --   makes the call and packs "results_buf".
    --
  is
    account_TDO_untyped:  System.untyped_word;
    account_TDO:  Object_Mgt.TDO_AD;
      FOR account_TDO USE AT account_TDO_untyped'address;

    account_one_untyped, account_two_untyped:
        System.untyped_word;
    account_one, account_two:
        Account_Mgt_Ex.account_AD;
      FOR account_one USE AT account_one_untyped'address;
      FOR account_two USE AT account_two_untyped'address;

    amount:  Long_Integer_Defs.long_integer;

    parameters, results:  buffer;
    FOR parameters USE AT param_buf;
    FOR results USE AT results_buf;


  begin
    case target_proc is
      when 0  => account_TDO_untyped := parameters.first_word;
                 Passive_Store_Mgt.Set_home_job(
                     tdo => account_TDO);
                 ADs_returned := false;

      when 1  => account_one_untyped := parameters.first_word;
                 amount :=
                     Account_Mgt_Ex.Get_balance(
                         account => account_one);
                 results := buffer'(
                     first_word  => System.null_word,
                       -- irrelevant
                     second_word => System.null_word,
                       -- irrelevant
```

```
                     amount      => amount);
                 ADs_returned := false;

      when 2  => account_one_untyped := parameters.first_word;
                 amount :=
                     Account_Mgt_Ex.Change_balance(
                        account  =>
                            account_one,
                        amount  =>
                            parameters.amount);
                 results := buffer'(
                     first_word  => System.null_word,
                       -- irrelevant.
                     second_word => System.null_word,
                       -- irrelevant.
                     amount      => amount);
                 ADs_returned := false;

      when 3  => account_one_untyped := parameters.first_word;
                 account_two_untyped := parameters.second_word;
                 Account_Mgt_Ex.Transfer(
                     source_account => account_one,
                     dest_account   => account_two,
                     amount         =>
                         parameters.amount);
                 results := buffer'(
                     first_word  => System.null_word,
                     second_word => System.null_word,
                     amount      =>
                         Long_Integer_Defs.zero);
                   -- irrelevant.
                 ADs_returned := false;

      when 4  => account_one_untyped := parameters.first_word;
                 Account_Mgt_Ex.Destroy_account(
                     account => account_one);
                 results := buffer'(
                   -- irrelevant.
                     first_word  => System.null_word,
                     second_word => System.null_word,
                     amount      =>
                         Long_Integer_Defs.zero);
                 ADs_returned := false;
      when others   =>
          RAISE System_Exceptions.operation_not_supported;
    end case;

    RETURN;

  end server_stub;

end Distr_Acct_Server_Stub_Ex;
```

X-A.7.23 `Distr_Acct_Init` Procedure

Initializes the distributed account manager globally for a distributed system.

```
with Account_Type_Name_Ex,     -- Example package.
     Attribute_Mgt,
     Authority_List_Mgt,
     Directory_Mgt,
     Job_Types,
     Identification_Mgt,
     Object_Mgt,
     Passive_Store_Mgt,
     Refuse_reset_active_version_Ex, -- Example package
     RPC_Mgt,
     System,
     System_Defs,
     Transaction_Mgt,
     Type_Name_Attribute_Ex,   -- Example package.
     User_Mgt,
     Unchecked_conversion;

procedure Distr_acct_init
  --
  -- Function:
  --   Initialization procedure for distributed
  --   account service.
  --     o Creates TDO.
  --     o Initializes and stores attributes.
  --     o Creates the service.
  --     o Creates  and installs the the server.
  --     o Stores and updates TDO, server, and service.
  --
  -- Logic:
  --   Private ADs are stored with pathnames and
  --   protected by authority lists. They are retrieved
  --   by the various modules that are part of the distributed
  --   account service at link-time.
  --
  -- History:
  --   06-02-88: Tobias Haas, Initial version.
  --
is
  use Transaction_Mgt;
    -- Import transaction operators.

  -- Pathnames:
  --
  account_name:  constant System_Defs.text :=
      System_Defs.text'(7, 7, "account");
    -- Pathname of account tdo.

  service_name:   constant System_Defs.text :=
      System_Defs.text'(15, 15, "account_service");
    -- Pathname of service AD.

  server_name:   constant System_Defs.text :=
      System_Defs.text'(14, 14, "account_server");
    -- Pathname of server job AD.

  -- Private ADs:
  --
  account_TDO:    constant Object_Mgt.TDO_AD :=
      Object_Mgt.Create_TDO;
    -- TDO for accounts.
  server:  constant  RPC_Mgt.RPC_server_AD :=
      RPC_Mgt.Create_RPC_server;
    -- Server for accounts.

  server_job:  Job_Types.job_AD;
    -- Installed server job.

  service:  RPC_Mgt.RPC_service_AD;
    -- Distributed service AD.

  -- Attribute-related stuff:
  --
  passive_store_impl:
```

```
      Passive_Store_Mgt.PSM_attributes_AD;
    -- Implementation of passive store attribute
    -- for accounts.

  type_name_impl:  System.untyped_word;
    -- Implementation of type name attribute
    -- for accounts.

  owner_only:  User_Mgt.protection_set(1);
    -- Protection set that includes only one ID, namely
    -- the type manager's owner.

  authority:  Authority_List_Mgt.authority_list_AD;
    -- Authority list that contains only one ID, namely
    -- the type manager's owner.

  type template is
    record
      dummy_word0:   System.untyped_word;
      is_homomorph:  boolean;
      dummy_word1:   System.untyped_word;
      dummy_word2:   System.untyped_word;
    end record;

    FOR template USE
    record AT mod 32;
      dummy_word0   at  0  range  0 .. 31;
      is_homomorph  at  4  range  0 .. 7;
      dummy_word1   at  8  range  0 .. 31;
      dummy_word2   at 12  range  0 .. 31;
    end record;

  type homomorph_AD is access template;
    pragma access_kind(homomorph_AD, AD);

  homomorph:  homomorph_AD;

  -- Auxiliary Stuff:
  --
  trans:  boolean := false;
    -- Set if local transaction is started.


  function Untyped_from_PSM_attributes is
      new Unchecked_conversion(
          source => Passive_Store_Mgt.PSM_attributes_AD,
          target => System.untyped_word);


  function Untyped_from_TDO is
      new Unchecked_conversion(
          source => Object_Mgt.TDO_AD,
          target => System.untyped_word);


  function Untyped_from_service is
      new Unchecked_conversion(
          source => RPC_Mgt.RPC_service_AD,
          target => System.untyped_word);

  function Untyped_from_homomorph is
      new Unchecked_conversion(
          source => homomorph_AD,
          target => System.untyped_word);

  function Untyped_from_job_AD is
      new Unchecked_conversion(
          source => Job_Types.job_AD,
          target => System.untyped_word);

begin
  -- 1. Allocate new passive store attribute implementation:
  --
  passive_store_impl := new
      Passive_Store_Mgt.PSM_attributes_object;
  -- 2. Allocate and initialize homomorph template:
  --
```

```
homomorph := new template'(
    dummy_word0  => System.null_word,
    is_homomorph => true,
    dummy_word1  => System.null_word,
    dummy_word2  => System.null_word);

-- 3. Initialize passive store attribute implementation:
--
passive_store_impl.homomorph := Untyped_from_homomorph(homomorph);

passive_store_impl.reset :=
    Refuse_reset_active_version_Ex.
        Refuse_reset_active_version'subprogram_value;

passive_store_impl.copy_permitted := false;

passive_store_impl.locking_area_start := 0;
passive_store_impl.locking_area_end := 0;
  -- Area in account where semaphore AD will be
  -- stored when account is activated.
-- 4. Store passive store attribute implementation with type:
--
Attribute_Mgt.Store_attribute_for_type(
    tdo       => account_TDO,
    attr_ID   => Passive_Store_Mgt.PSM_Attributes_ID,
    attr_impl => Untyped_from_PSM_attributes(
        passive_store_impl));
  -- Store PSM attribute.

-- 5. Initialize type name attribute implementation:
--
type_name_impl := Account_Type_Name_Ex'package_value;

-- 6. Store type name attribute implementation with type:
--
Attribute_Mgt.Store_attribute_for_type(
    tdo       => account_TDO,
    attr_ID   => Type_Name_Attribute_Ex.
                 Get_type_name_attr_ID,
    attr_impl => type_name_impl);

server := RPC_Mgt.Create_RPC_server;
-- 7. Install server:
--
server_job := RPC_Mgt.Install_RPC_server(
    server => server);

-- 8. Create the service:
--
service := RPC_Mgt.Create_RPC_service(
   server => server);


 -- 9. Create authority list to protect private ADs:
 --
 owner_only.length := 1;
 owner_only.entries(1).rights := User_Mgt.access_rights'(
     true, true, true);
 owner_only.entries(1).id := Identification_Mgt.Get_user_id;

 authority := Authority_List_Mgt.Create_authority(owner_only);

-- 10. Store and Update the TDO, attributes and service.
--     Use transactions to protect these operations:
--
if Transaction_Mgt.Get_default_transaction =
    null then
  Transaction_Mgt.Start_transaction;
  trans := true;
end if;

begin
  Directory_Mgt.Store(
      name   => account_name,
      object => Untyped_from_TDO(account_TDO),
      aut    => authority);

```

```
    Directory_Mgt.Store(
        name   => service_name,
        object => Untyped_from_service(service),
        aut    => authority);

    Directory_Mgt.Store(
        name   => server_name,
        object => Untyped_from_job_AD(server_job),
        aut    => authority);

    Passive_Store_Mgt.Request_update(
        Untyped_from_TDO(account_TDO));
    Passive_Store_Mgt.Request_update(
        Untyped_from_PSM_attributes(
            passive_store_impl));
    Passive_Store_Mgt.Request_update(
        type_name_impl);
    Passive_Store_Mgt.Request_update(
        Untyped_from_homomorph(homomorph));


    if trans then
      Transaction_Mgt.Commit_transaction;
    end if;
  exception
    when Directory_Mgt.entry_exists =>
      if trans then
        Transaction_Mgt.Abort_transaction;
      end if;

    when others =>
      if trans then
        Transaction_Mgt.Abort_transaction;
      end if;
      RAISE;

  end;

end Distr_acct_init;
```

X-A.7.24 `Distr_Acct_Home_Job_Ex` Procedure

Sets the home job of the account service.

```
with
  Distr_Acct_Server_Stub_Ex,
  Long_Integer_Defs,
  Passive_Store_Mgt,
  RPC_Call_Support,
  RPC_Mgt,
  System;


 procedure Distr_Acct_Home_Job_Ex is

   parameters, results: Distr_Acct_Server_Stub_Ex.buffer;
     -- Buffers for remote call.

   length:              System.ordinal;
     -- Gives actual length of results buffer in remote call.
     -- Not used here.

   service: constant RPC_Mgt.RPC_service_AD := null;
     pragma bind(service, "account_service");
     -- Account service. Retrieved at link-time.

   account_TDO_untyped: constant System.untyped_word
                        := System.null_word;
     pragma bind(account_TDO_untyped, "account");

 begin
   -- Set up server as home job
   --     by calling procedure ''0'':
   --
   parameters := Distr_Acct_Server_Stub_Ex.buffer'(
   first_word  => account_TDO_untyped,
     -- account TDO
   second_word => System.null_word, -- Irrelevant.
       amount      => Long_Integer_Defs.zero);
     -- Irrelevant.

   length := RPC_Call_Support.Remote_call(
       service        => service,
       target_proc    => 0,
         -- Server will call Passive_Store_Mgt.Set_home_job.
       param_buf      => parameters'address,
       param_length   => parameters'size,
       ADs_present    => true,
       results_buf    => results'address,
       results_length => results'size);

 end Distr_Acct_Home_Job_Ex;
```

X-A.7.25 `Makefile`

Makefile for the the preceding account type manager programs. To use type:

- make acct_active, or
- make non_xo, or
- make stored

to create different executable versions of the account type manager. **NOTE:** The distributed type manager is not yet implemented.

```
#Definitions:
lib = ada_library
impl = stored.b
messages = "(acct_mgt.s acct_vis.b acct_main.sb)"

spec_obj = $(lib)/acct_types.s.obj \
           $(lib)/conversion_support_ex.s.obj \
           $(lib)/account_mgt_ex.s.obj \
           $(lib)/acct_visual.s.obj

body_obj = $(lib)/acct_visual.b.obj \
           $(lib)/acct_main_loop.b.obj \
           $(lib)/account_mgt_ex.b.obj

tdo_spec_obj = $(lib)/type_name_attribute_ex.s.obj \
               $(lib)/account_type_name_ex.s.obj \
               $(lib)/refuse_reset_active_version_ex.s.obj

tdo_body_obj = $(lib)/type_name_attribute_ex.b.obj \
               $(lib)/account_type_name_ex.b.obj \
               $(lib)/refuse_reset_active_version_ex.b.obj

acct_active:  $(spec_obj) $(body_obj) acct_active_body
        link.ada acct_main_loop
        manage.program acct_main_loop $(messages)
        -mv acct_main_loop acct_active

non_xo: $(spec_obj) $(body_obj) non_xo_body  stored_account_tdo_init_ex
        stored_account_tdo_init_ex
        link.ada acct_main_loop
        manage.program acct_main_loop $(messages)
        -mv acct_main_loop non_xo

stored: $(spec_obj) $(body_obj) stored_body  stored_account_tdo_init_ex
        stored_account_tdo_init_ex
        link.ada acct_main_loop
        manage.program acct_main_loop $(messages)
        -mv acct_main_loop stored

acct_active_body: $(spec_obj) acct_active.b, account_mgt_ex.b.obj
        -ada acct_active.b

non_xo_body: $(spec_obj) non_xo.b, account_mgt_ex.b.obj
        -ada non_xo.b

stored_body: $(spec_obj) stored.b, account_mgt_ex.b.obj
        -ada stored.b

$(lib)/acct_visual.b.obj: $(spec_obj) \
                          acct_vis.b
        -ada acct_vis.b

$(lib)/acct_main_loop.b.obj: $(spec_obj) \
                             acct_main.sb
        -ada acct_main.sb

$(lib)/acct_visual.s.obj: $(lib)/acct_types.s.obj \
                          $(lib)/account_mgt_ex.s.obj \
                          acct_vis.s
        -ada acct_vis.s

$(lib)/acct_types.s.obj: $(lib)/account_mgt_ex.s.obj  \
```

```
                        acct_types.s
        -ada acct_types.s

$(lib)/account_mgt_ex.s.obj: acct_mgt.s
        pwd
        -ada acct_mgt.s

$(lib)/conversion_support_ex.s.obj: conv.s
        -ada conv.s

stored_account_tdo_init_ex: $(tdo_spec_obj) \
                            $(tdo_body_obj) \
                            type_name_attribute_init_ex \
                            acct_tdo.sb
        -ada acct_tdo.sb
        type_name_attribute_init_ex
        link.ada stored_account_tdo_init_ex

$(lib)/refuse_reset_active_version_ex.b.obj: $(tdo_spec_obj)
        -ada refuse_reset_av.b

$(lib)/type_name_attribute_ex.b.obj: $(tdo_spec_obj)
        -ada typnam.b

$(lib)/account_type_name_ex.b.obj: $(tdo_spec_obj)
        -ada actyna.b

$(lib)/refuse_reset_active_version_ex.s.obj: refuse_reset_av.s
        -ada refuse_reset_av.s

$(lib)/account_type_name_ex.s.obj: $(lib)/type_name_attribute_ex.s.obj \
                              actyna.s
        -ada actyna.s

$(lib)/type_name_attribute_ex.s.obj: typnam.s
        -ada typnam.s

type_name_attribute_init_ex: typnamattr.sb
        -ada typnamattr.sb
        link.ada type_name_attribute_init_ex
```

X-A.7.26 Named_copy_ex Procedure

```
with Directory_Mgt,
     Passive_Store_Mgt,
     System,
     System_Defs,
     System_Exceptions,
     Transaction_Mgt;

procedure Named_copy_ex(
    source:  System_Defs.text;
    dest:    System_Defs.text)
is
  --
  -- Function:
  --   Copies object tree at source pathname to
  --   destination pathname.  The source tree is the
  --   named source passive object and all passive
  --   objects reachable from it via successive
  --   master AD references.  The destination pathname
  --   must not already exist.
  --
  --   "Named_copy_ex" is transaction-oriented.
  --
  -- Exceptions:
  --   Directory_Mgt.entry_exists
  --   Directory_Mgt.name_too_long
  --   Directory_Mgt.no_access
  --   System_Exceptions.bad_parameter -
  --     Both the calling process and the
  --     destination directory have a
  --     null authority list.
  --   System_Exceptions.
  --       transaction_could_not_be_committed
  --
  -- Body:
  --   If there is no default transaction, then a local
  --   transaction is created and transaction timestamp
  --   conflicts are handled locally.  Any other
  --   exception is handled by aborting any local
  --   transaction and reraising the exception.
  --
  --   The root object AD is retrieved, a copy stub
  --   is created, the copy stub AD is stored under
  --   the destination pathname, and "Copy" is called
  --   to copy the tree.

  source_AD:  System.untyped_word;
  dest_AD:    System.untyped_word;
begin
  loop
    declare
      trans:  boolean := false;
        -- Set if local transaction is started.
      use Transaction_Mgt;
        -- Import "=" for "transaction_AD".
    begin
      if Transaction_Mgt.Get_default_transaction
          = null then
        Transaction_Mgt.Start_transaction;
        trans := true;
      end if;

        source_AD := Directory_Mgt.Retrieve(source);
        dest_AD   := Passive_Store_Mgt.
                       Create_copy_stub(source_AD);
        Directory_Mgt.Store(name   => dest,
                            object => dest_AD);
        Passive_Store_Mgt.Copy(source_AD, dest_AD);
      if trans then
        Transaction_Mgt.Commit_transaction;
      end if;
      RETURN;

    exception
      when System_Exceptions.
```

```
          transaction_timestamp_conflict =>
        if trans then
          Transaction_Mgt.Abort_transaction;
            -- Loop back and try again if
            -- transaction started locally.
        else
          RAISE;
            -- Reraise the exception if the
            -- transaction was already on the
            -- transaction stack.
        end if;

      when others =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        end if;
          -- Abort the transaction if it was
          -- started locally.
        RAISE;
          -- Reraise exception that invoked handler.

    end;
  end loop;

end Named_copy_ex;
```

X-A.7.27 `Older_than_ex` Function

```
with Long_Integer_Defs,
     Passive_Store_Mgt,
     System,
     System_Exceptions;

function Older_than_ex(
    a:  System.untyped_word;
    b:  System.untyped_word)
  return boolean
is
  --
  -- Function:
  --   Returns true if object "a"'s passive version is
  --   older than object "b"'s passive version.
  --
  -- Exceptions:
  --   System_Exceptions.bad_parameter -
  --     Either "a" or "b" does not have a passive
  --     version.

  use Long_Integer_Defs;
    -- Import "<" for long integers.

  a_info:  Passive_Store_Mgt.passive_object_info;
  b_info:  Passive_Store_Mgt.passive_object_info;
begin
  a_info := Passive_Store_Mgt.
              Request_passive_object_info(a);
  b_info := Passive_Store_Mgt.
              Request_passive_object_info(b);

  if not a_info.valid or else not b_info.valid then
    RAISE System_Exceptions.bad_parameter;

  else
    RETURN a_info.write_time < b_info.write_time;

  end if;
end Older_than_ex;
```

GLOSSARY B

This glossary defines important terms used in this manual. Some definitions apply to this manual and some apply to other parts of the BiiN™ system.

A

AD (access descriptor)
(1) A protected pointer to a system object. An AD identifies a particular object and includes *rights* that determine what operations are allowed on the object via the AD. An AD can also be null, referencing no object. (2) In Ada, one of the alternatives used by pragma `ACCESS_KIND`.

abort
Terminate a transaction unsuccessfully, reversing all changes associated with the transaction.

abstract data type
A data type with an unspecified representation. An abstract data type is defined entirely by its supported operations. OS object types such as files and directories are abstract data types.

access
Read or modify an object or datum.

access descriptor (AD)
A protected pointer to a system object. An AD identifies a particular object and includes *rights* that determine what operations are allowed on the object via the AD. An AD can also be null, referencing no object.

access method (AM)
A distinct way to use a device, defined by a set of I/O operations (typically `Open`, `Close`, `Read`, and `Write`). There are four access methods: byte-stream I/O, record I/O, character display I/O, and graphics display I/O. Each method is defined by a separate BiiN™ Ada package. Each device (pipe, file, directory, and so forth) supported by an access method has a different subset of the total operations available for the access method.

access rights
Bits in an access descriptor (AD) that restrict the sets of operations you can perform to manipulate an object. Access rights consist of three type rights bits (typically mapped to use, modify, and control for a particular service) and two representation rights bits (read and write). Type rights can be thought of as permissions granted to a caller by a service's type manager. The permissions allow the caller to perform certain operations on the type manager's objects. The representation rights bits are used only by type managers to read from and write to the representation of a particular type of system object.

access type

An Ada type consisting of pointers to values of a specified second type. Values of a particular access type are represented by either ADs, virtual addresses, linear addresses, or heap offsets. The `access_kind` pragma is used to specify the representation of an access type. Each access type also includes the special value `null`, indicating a pointer to nothing. If an access type is represented with ADs then referenced values are represented by system objects.

action

(1) A record that specifies an event to be signaled, a destination to which the event is signaled, and an optional two-word message to all receivers of the event. A valid destination is a process, a job, or an event cluster. (2) In SMS, the user-defined command to be executed when a condition on a target is satisfied. The possible actions for an SMS event include sending a `mail` message to the subscriber, broadcasting a message, or executing a BiiN™ CL command script in a batch session.

activation model

A characteristic of an object type that specifies how objects of the type are activated. The multiple activation model activates an object in any job or node. The single activation model activates an object only in a particular home job (for local objects) or home node (for global objects); another job or node that attempts to activate the object instead activates a *homomorph*, a token object that stands in place of the actual object.

activate

To create an active version of a system object from its current passive version. Objects are activated automatically when there is no active version of an object and a program references the object's representation. Activating an object activates ADs in the object but does not activate referenced objects.

active memory

The virtual memory of a particular BiiN™ node, as distinct from the passive store of a distributed BiiN™ system.

active AD

An AD in active memory, represented by one memory word.

active object

A system object in active memory.

active version

An active object that has been activated from a passive version. An object can have multiple active versions, in different jobs or at different nodes.

active-only object

An object that does not and cannot have a passive version. An object's type determines whether or not it can be passivated.

actual parameter

Value or variable supplied as a parameter in a specific invocation of a call.

Ada

A standard programming language for programming large-scale and real-time systems. BiiN™ Ada is a complete implementation of Ada as specified by ANSI/MIL-STD-1815A, 22 January 1983. The BiiN™ Ada implementation adds implementation-defined pragmas and attributes as the standard allows.

address

A value that can be used to access a particular object or memory location. An address may be an AD, virtual address, linear address, or physical address. Physical addresses are only used by the hardware and inside the OS.

address space

A set of memory locations. Each location is an <address, value> tuple. Address spaces include object address spaces, virtual address spaces, linear address spaces, and physical address spaces.

address translation

The process of converting a linear address or virtual address to a physical address. Address translation may trigger paging or object activation to load needed information into physical memory.

advisory parameter

A parameter that advises a service but does not dictate its actions. A service may ignore an advisory parameter or substitute a different value.

aggregate

(1) An Ada composite value, of an array or record type, consisting of element values listed within parentheses. (2) In C, an array, structure, or union.

age factor

The rate at which a waiting job ages in the scheduler's waiting queue (regardless of priority or service level). On every scan of the waiting queue, the age factor is added to the job's age to determine a new age. The larger the aging factor, the faster a job ages, and the sooner it rises to the front of the waiting queue.

alias

(1) In general, an entity that stands for another entity. (2) In the BiiN™ OS, a non-master passive AD. (3) In BiiN™ C, an identifier that is defined with the `#pragma alias` preprocessor control and is used to associate an identifier with its external definition. This type of alias is needed to refer to functions or data implemented in other languages with different forms for identifiers. (4) In the BiiN™ Systems Object Module Format, a two-byte number used as an abbreviation for a symbolic name in a single object module. (5) In CLEX, a short command that stands for a longer command.

alias AD

A passive AD that is not a master AD. An alias AD can refer to an object stored on a different volume set than the AD itself.

amplify

Add rights to an AD to some object. Amplifying rights is a privileged operation, requiring an AD to the object's TDO, with amplify rights.

amplify rights
A type right for TDOs, required to amplify rights on ADs.

argument
(1) Values specified as part of a command. Arguments are defined with the `manage.commands` utility. An argument may be *mandatory* or *optional*. An argument has a name (prefixed by a colon: `:argument_name`), a type (one of: *boolean*, *integer*, *pointer*, *range*, *string*, *string list*, or *derived*), and a value ([=some_value]). Optional arguments may have a default value. Arguments may be entered in *named* or *positional* notation. (2) An expression that appears within the parentheses of a subprogram call. The expression is evaluated and the result is copied into the corresponding parameter of the called function.

array type
A structured data type consisting of a fixed number of components or elements, which are all of the same type.

ASCII (American Standard Code for Information Interchange)
A standard seven-bit code representing alphabetic, numeric, punctuation, mathematical, and control characters.

atomic operation
An operation that always succeeds completely or fails completely. An atomic operation never produces partial output or partial changes in its environment before failing. An atomic operation may also acquire locks to ensure that intermediate results are not visible to concurrent operations.

attribute
(1) A property that can be associated with multiple system objects or object types. (2) A language-defined characteristic of a named Ada entity, such as `'size` or `'image`. Some Ada attributes are functions.

attribute call
A subprogram invocation where the module implementation used is selected at invocation-time, based on the object type of the invocation's first actual parameter.

attribute entry
An <attribute ID, attribute value> tuple that gives an attribute's value for a particular system object or object type.

attribute ID
A system object that identifies an attribute.

attribute instance
An attribute value stored in a particular TDO.

attribute list
A system object that contains a list of object-specific attribute entries, for a particular object.

attribute package
A package that has different implementations for different system object types or system

objects. For example, `Byte_Stream_AM.Ops` is an attribute package. An attribute package can only contain subprograms.

authority list
List of IDs and associated type rights. An authority list is associated with an object, and a caller must hold an ID that matches one in the authority list, with the appropriate rights, before the caller can access that object.

B

backup service
The OS service that manages backup and restore operations.

base priority
The lowest priority a process can have. It is determined initially by the SSO priority of its job (for a job's initial process) or by the base priority of its parent (for a spawned process). It may be changed by the user or the system administrator.

basic disk
A device that supports low-level access to a disk as an array of sectors or bytes via record I/O or byte stream I/O.

basic I/O service
The OS service that manages byte stream I/O, standard Ada I/O, and common I/O definitions.

basic streamer
A device that supports low-level access to a streamer tape via record I/O and byte stream I/O.

batch job
A job that consists of a batch of requests (a background job with no attached user). Like interactive jobs, batch jobs run in normal memory, have limited processor claim, and have a lower priority than real-time and time-critical jobs.

bi-paged object
An object representation in which the object is so large that its page table must also be paged. A bi-paged object's size ranges from 8M bytes to 4G bytes.

body
A BiiN™ Ada program unit containing the declarations and statements that implement a package, subprogram, or task specification.

byte stream I/O
An I/O access method that provides data transfer as an uninterpreted stream of bytes. Some implementations support random access to particular byte positions in the stream.

blocked
State of a process that is unable to execute because it is waiting on an event, a port, or a semaphore.

Boolean
(1) Either true or false. (2) In BiiN™ Pascal, a predefined type.

built-in commands
Commands built into BiiN™ CL, part of all command sets. Built-in commands entered to CLEX or to a program are executed by the command service itself.

byte
A unit of memory containing eight bits and aligned at an 8-bit boundary. Each byte has a distinct address, whether linear, virtual, or physical addresses are used. Bits in a byte are numbered from 0 to 7.

C

call
(1) A subprogram. (2) A particular invocation of a subprogram. (3) To invoke a subprogram.

central system
Central part of a BiiN™ node, containing one or more P7 GDPs, one or more system buses, and shared memory.

Channel Processor (CP)
A P7 component that handles I/O transfers between a BiiN™ node's central system and I/O subsystems. The CP is the main hardware component of an I/O module.

character display I/O
An interactive access method that provides operations on character display terminals. Character display I/O is defined by the `Character_Display_AM` package. Character display I/O can also be used for output to printers.

character display device
A device that displays and manipulates ASCII characters on a two-dimensional surface. Typical examples are printers and windows on terminal screens; typical operations on such devices include input, output, cursor movement, manipulation of the display surface, control and status activities, and identifying and changing the attributes associated with a device.

character terminal
A terminal that has some subset of the features specified in the *ANSI X3.64* standard; for example, character insertion and deletion, line insertion and deletion, cursor positioning, and scrolling. The *DEC VT-100* (a trademark of Digital Equipment Corp.) is a typical character terminal.

character terminal manager
A device manager that supports access to character terminals.

character terminal user agent
Software that allows a user to control the windows on a character terminal. It is provided by the character terminal manager.

child process
A process that is created (spawned) by another process (called the parent process).

CL (Command Language)
The BiiN™ command language, used for invoking and controlling the execution of programs and scripts. CL is implemented by the command service.

Clearinghouse
A BiiN™ database that keeps track of where objects and IDs are within an entire distributed system. While objects and IDs are actually stored on physical nodes, the Clearinghouse keeps track of which node houses which objects and IDs.

clearinghouse service
The OS service that provides packages to manage the Clearinghouse to store names and node addresses across a distributed system.

CLEX (Command Language Executive)
The BiiN™ command interpreter of BiiN™ CL commands. CLEX is used for invoking and controlling the execution of programs and BiiN™ CL scripts.

cluster
Group of I/O queues that are serviced together. A cluster represents a group of devices, typically those serviced by the same CP task.

clustered file
A structured file whose records are organized in related groups ("clusters") according to a clustering b-tree organization index.

command
(1) A directive to a program (including *CLEX* itself) or script. A command consists of a *command name* followed by command *arguments* or *control options*. An *invocation command* is given to *CLEX* to invoke a program or BiiN™ CL script. *Runtime commands* are entered to control the operation of a program or BiiN™ CL script. *Built-in commands* are part of the command language (BiiN™ CL) itself. Commands are processed either by CLEX (*CLEX commands* and *invocation commands*), or by the Command Handler (*built-in commands* and *runtime commands*). (2) In mass storage I/O modules, a command defines the operation to be performed by the I/O Module.

command history
A record of all entered commands. There are several *built-in commands* provided by the command service to create, list, and re-execute a command history (a *history log file*). There is also a *control option*, `::history`, which creates a history log file for the given command.

command name
A sequence of characters, such as `create.alias`, that identifies a BiiN™ CL command. The command name is the first part of a complete *command*. There may be two parts in a command name, the verb (`create`) and the noun (`alias`), separated by a period. Command names may be shortened to the minimum unique abbreviation (`c.al`).

command script
A file containing a sequence of BiiN™ CL commands that are interpreted by CLEX. A command script differs from a command file in two important ways: (1) You can pass arguments to a command script, but not to a command file. (2) The command script is

interpreted as a separate job, whereas a command file is executed in the program's environment.

command service
The service that parses and returns commands for programs (including CLEX itself) and BiiN™ CL scripts. Built-in commands are processed by the command service itself.

command set
A command set defines the *runtime commands* currently available. A program using the command service always has at least one command set. All command sets include the BiiN™ CL *built-in commands.*

commit
Complete a transaction successfully. If the transaction is not contained in some other transaction, then any changes associated with the transaction are made permanent.

compaction
A memory management daemon that relocates system objects and other memory segments to reduce fragmentation of normal memory. Compaction is transparent to application software.

compilation unit
(1) In general, a building block of a program or subsystem that, when compiled, produces a single object module. (2) When using the BiiN™ Application Debugger, a single unit of compilation, defined differently for each BiiN™ language and corresponding to a single object module. Referred to as a CU. (3) In BiiN™ Ada a specification or body of BiiN™ Ada package, subprogram, or task, presented for compilation as an independent text. (4) In BiiN™ C, any primary source file (excluding those that are "included").

compiler
A system program that translates high-level language source files into one or more object modules (contained in one or more object module files, depending on the language).

concurrent
Happening at the same time.

concurrent program
A program divided into pieces that appear to execute simultaneously.

concurrent programming service
The OS service that supports concurrent programs, programs with multiple processes and jobs executing together.

configurable object
A representation of a hardware or software component of a BiiN™ node that must be configured at node initialization, or can be dynamically added to a running system.

configuration service
The OS service that manages configuration of a BiiN™ node.

consistency level
Within transactions, the level of interference a transaction can tolerate within a file. A transaction can have level 1, level 2 or level 3 consistency.

constant
A value that does not change; can be either symbolic (named) or literal.

constraint
(1) BiiN™ Ada restriction on the set of possible values of a type or subtype. A range constraint specifies lower and upper bounds on the values of a scalar type. An accuracy constraint specifies the relative or absolute error bound on values of a real type. An index constraint specifies lower and upper bounds on an array index. A discriminant constraint specifies particular values of the discriminants of a record type or private type. (2) In BiiN™ SQL, a restriction on the set of possible values that may be stored in a column.

constraint_error exception
BiiN™ Ada built-in exception raised by the BiiN™ Operating System or the BiiN™ Ada runtime system when a runtime constraint is violated. Common causes of `constraint_error` are (a) a value that violates a constraint in an assignment statement or subprogram call; or (b) a null access descriptor parameter.

control option
A predefined directive to a command that modifies the execution behavior or the I/O behavior of the command. A control option consists of a name (prefixed by a double colon, `::control_option`), and a value (`[=]value`).

control rights
One of three type rights. By convention, control rights are required to destroy or restructure an object.

countable global object
A global object that exists so long as any job may be using it. ADs to countable global objects are local ADs; such ADs cannot be stored in global objects.

create right
A type rights for TDOs and SROs. Creating an object requires create rights on the new object's TDO and on the SRO used to allocate the object.

CRP
The *current record pointer* represents the current location in a *structured file*.

current directory
Current location in a directory structure. If a relative pathname is specified, names are looked up starting from this directory. The current directory is always stored in process globals.

current record pointer
See CRP.

cursor
(1) In BiiN™ SQL, a named query. The cursor mechanism itself is a pointer that provides

row by row access to the result table produced by the query. The cursor can be moved with FETCH or FETCH BACK. (2) A special marker that identifies specific cells within a frame buffer. For example, a *write* operation might write characters at a cursor's current location and then move the cursor to a new location.

D

daemon
A server process that provides a service asynchronously. For example, daemons service spool queues, batch queues, and timed request queues. Memory management daemons provide compaction, and garbage collection.

data abstraction
The design principle that data representation should be concealed from users of a data type, and that data should be defined to users in terms of its behavior, not its representation.

data area
A set of disk space allocations on a single volume set. The primary data area contains the file's actual data. Secondary data areas are used to allocate space for indexes.

data definition service
The OS service that manages data definitions.

deadlock
A situation that occurs when two or more processes are blocked and each process is waiting for resources or signals controlled by other blocked processes.

debug object (DO)
The (internal) part of a domain that contains the symbolic debug information for the domain. A debug object is composed of one or more debug units.

deallocate
Destroy an object's representation in active memory. If the object has a passive version, then its active version can later be recreated.

declaration
A program construct that associates a name with a program entity, such as a type, constant, variable, or subprogram.

default
Value used for an actual parameter if no value is specified in the invocation.

default transaction
Transaction (if any) at the top of a process's transaction stack. The default transaction is usually the most recent transaction started by the process. Transaction operations use the caller's default transaction if no transaction is explicitly specified.

default value
A value assigned to a formal parameter when the corresponding actual parameter is omitted.

delete
An operation used to remove a record, directory, character, object, or other entity.

derived
(1) In BiiN™ CL, an **argument** type. A derived argument's type is *derived* from the value's representation. A value of `true` or `false`, or just an argument name, implies a *boolean*; a series of digits implies an *integer*; a double period, optionally with an integer on either side, implies a *range*; a value in quotation marks is considered a *string*; string values in parentheses imply a *string list*. (2) A category of data types supported in BiiN™ C: arrays, pointers, structures, and unions.

device
Physical or logical entity that supports one or more access methods.

device class
A specification that defines the devcie-specific details necessary to access a member of a class of devices using an I/O mechanism.

device driver service
The OS service that supports device drivers.

device manager
Module that implements all operations on a particular device type. Implementations of each access method supported by the device type are part of the device manager.

Device Services
The OS service area that provides support to write and use device drivers.

directory
System object that associates names (entry names) with non-null ADs. A directory is the main way to associate a name with the AD's underlying object.

directory entry
A <name, AD> pair stored in a directory.

directory name
Part of a pathname that names the directory containing the named entry.

Directory Services
The OS service area that supports associating names with objects, protecting objects stored in directories, and retrieving objects based on a given name.

discrete type
A BiiN™ Ada enumeration type or integer type. Discrete types are used for array indexing, for loop iteration variables, and for choices in `case` statements and record variants.

discriminant
Record component that can determine the subtype of, or the presence or absence of, other record components.

discriminant constraint
Constraint on a record subtype that specifies a value for each discriminant of the record type.

disk volume label
A printable name assigned when a disk volume is logically initialized. This name is stored on the disk volume and does not have to be unique.

dispatch
Bind a ready process to an available General Data Processor (GDP) for execution.

dispatching mix
The set of jobs that are eligible for execution on a node. All processes in a job move in and out of the dispatching mix together, under control of the BiiN™ Operating System scheduler. A process can be blocked or suspended for other reasons while it is in the dispatching mix.

dispatching port
System object at which ready processes are queued to be dispatched and executed by P7 GDPs.

distributed
Property of a service that can be transparently accessed from different nodes in a BiiN™ distributed system.

distributed service
A service that can be transparently accessed from different nodes in a BiiN™ distributed system. For example, the object service, transaction service, naming service, and filing service are distributed services.

distributed system
A collection of hardware systems (*nodes*) connected by networks and sharing a common clearinghouse and one figurehead naming domain. The operating system unifies all the nodes into a single system, by providing distributed services that make data and resources accessible from any node.

domain
In architectural terms, a domain object, its associated linear address space, and software-predefined system objects.

domain object
A system object that defines and protects an execution environment.

E

elaboration
(1) Execution of a declaration in a BiiN™ Ada program unit or block. Elaboration executes any initialization code for variables or packages elaborated. (2) In BiiN™ Ada, the elaboration of a declaration is the process by which the declaration achieves its effect (such as creating an object); this process occurs during program execution. (3) When using the BiiN™ Application Debugger, the process by which program entities come into existence at run time. For example, the elaboration of a variable declaration involves allocating memory for a variable. A program entity cannot be accessed by the debugger until it has been elaborated.

embedded object

An object representation that is contained entirely in the object's descriptor. Only zero-length objects and semaphores use embedded representations.

emulation

An object that interprets higher-level printing functions for a printer and produces the expected output by simulating the function using more primitive functions available on the target printer.

enumerated

In BiiN™ CL, an argument subtype (of type string). An enumerated value has a defined set of allowable string values; for example, `"start"`, `"middle"`, `"end"`.

enumeration type

Discrete type with values listed in the type declaration. Values of an enumeration type can be identifiers or (in BiiN™ Ada) character literals.

error

(1) One of the levels of diagnostics generated by the BiiN™ Ada, C, FORTRAN, COBOL, and Pascal compilers and the BiiN™ Systems Linker. Errors are conditions that may affect the generated output, but from which the compiler or linker can recover (by ignoring an operand or operation, modifying or ignoring a statement, and so on). Processing continues and output can be generated. However, the output may no longer do what you intended. (2) One of the exit codes provided by the BiiN™ Ada, C, FORTRAN, and Pascal compilers and the BiiN™ Systems Linker. This exit code indicates that one or more error or serious error diagnostics were issued.

event

(1) An indication of the occurrence of some activity within the system that concerns a process or group of processes. Events are local or global depending on the scope of their effect. (2) In SMS, a change in state of some object that is of interest to a user. An SMS event consists of a target, a condition and an action.

event cluster

System object that groups up to 32 events. Each process and job has its own associated event cluster. Programs can create additional event clusters and associate processes with them.

event handler

A procedure executed asynchronously in response to an event. Handler execution interrupts normal execution of the process that receives the event.

environment variable

Another name for a BiiN™ CL variable, especially those variables that control the behavior of an executing program.

exception

(1) In general, an error condition. (2) A BiiN™ Ada-defined error indication. To raise an exception transfers control to an exception handler. If the current block or call does not contain a handler for a raised exception, then the exception is propagated to the calling block or call, which may handle the exception or propagate it further. (3) A run-time condition

that may cause the output of a program to be wrong due to an algorithmic mistake in the source program or due to invalid input; also called a run-time error. The term exception implies that, in some cases, a routine can be called to handle the situation, and then processing can continue normally. (4) Raised by BiiN™ SQL procedures that are called by BiiN™ Ada procedures as an alternative to the standard SQLCODE parameter.

exception handler
A sequence of statements executed in response to an exception. Known as a trap handler in FORTRAN.

executable program
A collection of software modules that has been linked (using the BiiN™ Systems Linker) and is ready for execution on a BiiN™ system. An executable program must have a main entry point and should (but need not) have all of its symbolic references resolved.

execute
(1) To perform machine instructions. (2) To perform an I/O Module operation.

execution environment
Consists of a linear address space partitioned into four regions (static data, instruction, stack, and operating system-reserved), a set of global and floating-point registers, an instruction pointer, and an arithmetic control register.

F

fault
A processor-detected error during program execution. For example, if an addition operation overflows, the GDP detects the error and raises a fault, which is handled by the BiiN™ Operating System as an exception.

fault tolerant
Property of a hardware configuration that lets it continue operating after a component failure without losing or corrupting data or programs.

field
(1) In Pascal, a component or element of a record type. (2) In the BiiN™ operating system, a contiguous portion of a record that is an instance of a single data item.

file
(1) A collection of information on a physical input or output device. (2) A system object that stores data on disk, organized for efficient random access, reading, and writing. Files cannot contain access descriptors. Files support byte-stream I/O and record I/O. (3) In BiiN™ Pascal, a predefined type.

file organization
Data structure used for a file; one of: stream, sequential, clustered, hashed, unordered, and relative.

filing volume set
A volume set providing external storage space for files and objects.

filing service
The OS service that manages files and records.

floating-point type
A numeric data type that represents numbers using exponential notation: $f*2**e$ (where f is a positive or negative fraction, normally in the range: $0.5 <= |f| < 1.0$; and e is a signed integer). Floating-point numbers can represent a wide range of numbers, but with incomplete precision. They are called "floating point" because the radix point "floats" based on the varying exponent, instead of being determined by a fixed scale factor determined by the data type.

form
A displayable, interactive document with labels and spaces for entering data.

formal parameter
A parameter as viewed within the subprogram it is a parameter for. A formal parameter has a name, a type, and a mode. Each subprogram invocation associates a different actual parameter with each formal parameter.

form description
A DDef that describes the physical layout and interactive capabilities of a form.

form service
The OS service that manages forms.

fragmentation
The division of free storage into multiple non-contiguous segments, caused by the normal operation of heap allocation, deallocation, and garbage collection.

frame buffer
The drawing space of a virtual terminal. An application writes to the frame buffer associated with a virtual terminal. Part of the frame buffer is visible to a user through a window; this visible part is called a view.

frozen memory
Memory for system objects that are never swapped out to disk and never relocated by compaction. Contrast with *normal memory*. Frozen objects can be accessed without page faults or delays due to compaction. However, resizing a frozen object may make it inaccessible during the resize operation.

full pathname
A pathname with three leading slashes. The BiiN™ OS evaluates full pathnames by first discovering which node to begin from, which may require a call to the Clearinghouse.

function
(1) A BiiN™ Ada, FORTRAN, or Pascal subprogram that returns a value to its caller. (2) The primary unit from which C language programs are constructed. Functions need not return a value to the caller. All C functions are external; that is, a function cannot contain another function. (3) In BiiN™ SQL, one of a set of five "built-in" functions that take the rows in a table or the set of values in a column as an argument (MIN, MAX, SUM, AVG, COUNT).

G

G
2^{30} = 1,073,741,824. For example, 1G bytes equals 1,073,741,824 bytes.

garbage collection
The process of identifying and reclaiming active objects that can no longer be accessed. Garbage collection reclaims both memory and object descriptors for reuse. Garbage collection is asynchronous and transparent to applications software. A global garbage collector reclaims global garbage and runs at every node, under administrative control. A local garbage collector is configured in a job if the running program requests it.

garbage object
An active object that cannot be accessed because it cannot be reached via active ADs. A garbage object can be reclaimed by garbage collection.

generic object
An object used as just a memory segment. A generic object does not have a type manager and all generic objects have the same TDO.

generic package
An Ada template for a package. Such a template can be instantiated with parameters at compile-time to create a package.

generic subprogram
An Ada template for a subprogram. Such a template can be instantiated with parameters at compile-time to create a subprogram.

generic unit
BiiN™ Ada template for a set of packages or subprograms. A package or subprogram created using the template is an instance of the generic unit. A generic instantiation is the kind of declaration that creates an instance. A generic unit is written as a package or subprogram specification prefixed by a generic formal part that may declare generic formal parameters. A generic formal parameter is either a type, subprogram, variable, or constant.

global
(1) An object or entity that is not local to a particular job. **(2)** A program-defined entity, such as a type, constant, or variable, that is declared outside a particular subprogram.

global AD
An AD that can be stored in a global object. A global AD's local bit is zero. A global AD normally references a global object.

global debug table (GDT)
A table of compilation units and their addresses generated by the BiiN™ Systems Linker.

global garbage collector
A memory management daemon that reclaims global garbage at a node. The global garbage collector is invisible to applications software. A system administrator controls a node's global garbage collector.

global memory
The collection of global objects in a node's active memory, combined with the free global memory available in the node's global SROs.

global object
A system object that exists outside of any particular job. A global object may be a countable global object or unbounded global object.

global SRO
An SRO used to allocate global objects. A node's active memory contains two global SROs, the normal global SRO and the frozen global SRO.

global variable
Global variables exist for the duration of a session. Variables created or modified by a program are local to the creating job, unless specified as global. Global variables are inherited by subsequent jobs in the same session.

H

handler
Code that is invoked by the BiiN™ Operating System or a language run-time system in response to an asynchronous occurrence rather than an application call. A handler can be an event handler, exception handler, or interrupt handler.

handler object
The handler object is a compiler-defined object that contains a table of the exception handlers defined in a domain. It is used by the compiler's runtime system to find the correct handler for a given exception.

hashed+file
A *structured file* whose records are organized according to a hashed organization index.

HDLC service
The OS service that manages High-Level Data Link Control communication.

head object
The initiating member of a pair of configurable objects associated with each other. A head object is characterized by its ability to function normally without being attached to another configurable object.

high-level scheduling
Putting a job in the hardware dispatching mix. When a job is invoked, it is enqueued on a scheduling port served by a scheduling daemon. When the daemon is activated, it removes the job from the port and schedules it by enqueueing the job's initial process at the end of one of the queues in a dispatching port. The port has 32 queues, ordered in priority from 0 (lowest) to 31 (highest). A process enqueued in this manner is said to be *in the mix.*

history
A record of occurrences.

history log file
A file of commands entered, and messages written, for a given job, session, or command. See *command history.*

home directory

Directory in which a user is placed after a successful login. The home directory is typically the highest directory owned by the user. All other stored objects owned by the user are normally subordinate to the home directory.

home node

The node at which a stored object's home volume set is currently mounted.

home volume set

The volume set that contains a stored object's passive version.

homomorph

An active version created as a token in place of a single-activation object that is only activated in a different home job or home node. The object's type manager must communicate with its counterpart at the home job or home node in order to access the object. Users of an object, outside its type manager, cannot distinguish between a homomorph and the object that it stands in place of.

Human Interface Services

The OS service area that provides integrated packages for quickly developing applications. All services in this area are based on a data definition (*DDef*) that supports the idea of building complex structures from small pieces (forms and reports), and that might be used to create informational output.

I

ID

(1) A system object that represents a particular class of access to a BiiN™ system. Each user is represented by an ID. Each group of users that share access to particular objects can be represented by an ID. The "world" class, denoting access granted to arbitrary other users, is represented by an ID. Application programs and type managers can use IDs to restrict access to stored objects to only certain programs or modules. (2) An index that identifies the device or controller to which an I/O module command/operation is directed.

ID list

A system object that contains a list of IDs. Each process has an associated ID list, referenced by its process globals, used for authority list evaluation in retrieving stored ADs protected by authority lists.

I/O message

A data transfer mechanism that is composed of four parts: a common, fixed part, a part for the exclusive use of a device driver and I/O processor, a part for the exclusive use of a device manager, and an array of buffer descriptions.

I/O shared queues

A data transfer mechanism employing an input and output queue per device. Designed for low-speed, character-oriented I/O, such as character terminals and printers.

I/O Services

The OS service area that supports all input/output to and from files and devices.

image module

An independently linked, protected, and potentially shareable piece of software that is bound to a program at runtime. Image modules support runtime linking, protection, and sharing.

incident

A BiiN™ construct that assigns a unique identifier, an *incident code*, to each error or exceptional situtation. An incident code references a message file, an individual message within that file, and a severity level.

incident code

Representation of a software incident. An incident code indicates the module which defines the incident, the incident number within the module, the incident severity, and a pointer to a message file.

index

The mechanism in which a data value is presented to an ordered list that contains the location of the desired value in a file. The index does not often contain all the values of the data item, but simply a limiting range of values. An organization index is an index for a clustered file or hashed file that influences the placement of records in the primary data area. An alternate index is an index in a structured file that in no way influences the placement of records in the primary data area.

index constraint

A restriction on a BiiN™ Ada array type or subtype that specifies the lower and upper bounds (and thus the number of values) for each index (subscript) of the array.

index type

The type of the array selector or index that is used to reference an element of a Pascal array. A Pascal index type must be an ordinal type.

initial_age

A job's age when it first enters the scheduler's waiting queue of swapped-out jobs. Larger values indicate older jobs. The job at the head of the queue is the oldest job and is scheduled for execution before the other jobs in the queue.

input event

An action performed by the user when interacting with an appliation through a terminal window. Typical examples are *mouse* and *keyboard* input events. Input events are forwarded to the application.

input focus

The virtual terminal to which a physical terminal's keyboard and mouse input are connected at a given time.

instance

Member of a class. For example, an instance of an attribute, an instance of a generic package.

instantiation

Operation performed by the BiiN™ Ada compiler to create an instance of a generic package or subprogram.

instruction object
The predefined system object that contains the code belonging to a particular domain. This object represents the instruction region, region 1.

integer
(1) An exact representation of a positive, negative, or zero value. (2) In BiiN™ CL, an argument or variable type. (3) One of the data types of BiiN™ FORTRAN. In BiiN™ FORTRAN, an integer datum can occupy 1, 2, 4, or 8 bytes; the default is 2 or 4, depending on the value of the compiler's `:intsize` argument. (4) In standard Pascal, a sequence of decimal digits. In BiiN™ Pascal, a sequence of binary, octal, decimal, or hexadecimal digits.

integer type
(1) Any type containing only whole numbers in a particular range. (2) One of the C-language data types `char` or `int` (all sizes, signed or unsigned). (3) One of the Pascal data types: `char` or `integer`.

interactive job
A job that interacts with a human user. Interactive jobs run in normal memory, have limited processor claim, and have a lower priority than real-time and time-critical jobs.

interrupt
Asynchronous hardware signal indicating some occurrence (such as I/O) that requires action by an I/O module.

interrupt handler
A procedure invoked in response to an interrupt.

interrupt reply procedure
A subprogram specified by a device manager in an I/O message that enables a device manager to process the reply information contained in an I/O message that has been serviced by either an I/O processor or a device driver.

invocation command
A BiiN™ CL command that invokes (calls and starts) a program or BiiN™ CL script.

J

job
A system object that represents an executing program. Each job has its own storage resource and its own address space. Each job has its own processing resources; scheduling for a node is done on a per-job basis. Resource control and reclamation is done on a per-job basis. A job can contain multiple processes executing concurrently.

K

K
2^{10} = 1,024. For example, 1K bytes equals 1,024 bytes.

key
A value used to designate a data item in a *record.* A *primary key* is a key value that uniquely identifies a record in a file. A key value that does not uniquely identify a record in a file is a *secondary key.*

kidnapped process
Process interrupted by an interrupt handler. The process is restored to its prior state and resumes execution when the handler completes.

L

LAN service
The OS service that manages Local Area Network communication.

library unit
A compilation unit that is not a subunit of another unit. Library units belong to a program library.

lifetime
A system object characteristic that determines how long an object can exist and how the object can be deallocated. There are three possible lifetimes: local, countable global, and unbounded global. Local objects are local to a job, exist no longer than their job, and can be deallocated by job termination or a local garbage collector. Countable global objects are shared by one or more jobs and can be deallocated when the jobs are no longer using the objects. Unbounded global objects have an unbounded lifetime and can be reclaimed by global garbage collection when the objects are no longer accessible via any AD.

lifetime check
A check, whenever an AD is copied, to ensure that a local AD is not copied into a global object. Attempting such a copy raises
`System_Exceptions.lifetime_violation`.

limited type
A BiiN™ Ada type that does not allow assignment or comparisons for equality.

linear address
A word interpreted as a 32-bit ordinal that specifies a byte offset into a linear address space. Bits 30 and 31 specify one of four region objects. Bits 0-29 specify a byte offset into the selected region. Region 0 contains static data. Region 1 contains instructions. Region 2 is a stack. Region 3 is used by the OS and is identical for all linear address spaces at a particular node.

linear address space
A 2^{32} byte (4G byte) address space partitioned into four regions, defined by a domain and a particular process. A domain contains ADs for region 0 (static data object) and region 1 (instruction object). A domain contains a subsystem ID that determines which of a process's stacks is used as region 2. Region 3 is defined by the OS and never changes. The linear address space contains holes where region objects are less than 1G byte in size.

link object
A system object with an system object type that supports the BiiN™ Operating System link attribute. When an AD for a link object is retrieved from a directory, an associated link evaluation function is called to evaluate the link and return a different AD. For example, a symbolic link system object contains a pathname. Retrieving an AD for a symbolic link triggers the retrieval of the AD named by the pathname in the symbolic link object.

linker
The BiiN™ software tool that combines the object modules created by the BiiN™ Ada, C, FORTRAN, COBOL, Pascal, and SQL compilers with the languages and systems environment to build an executable program. Besides producing the executable program directly from the object modules created by compilers, the linker can also produce image modules from object modules.

literal
(1) A symbol or number that represents a specific value rather than naming a value defined elsewhere (variable or constant) or describing a computation (expression). A literal can be a numeric literal, enumeration literal, character literal, or string literal. (2) In BiiN™ SQL, the representation of character strings, exact numeric values (FIXED) and approximate numeric values (FLOAT).

lock
An entity that allows a *transaction* or *opened device* to ensure that it alone has access to a particular resource.

local
(1) An object or entity that is local to a particular job. (2) A scope of an entity, such as a constant or variable, that is declared and visible only within a particular subprogram or block.

local AD
An AD that is local to a job. A local AD cannot be contained in or copied to a global object.

local bit
A bit in an AD that is one in a local AD and zero in a global AD. The local bit is not interpreted in null ADs.

local garbage collector
A memory management daemon that reclaims local garbage within a job. A running program must request local garbage collection or else no daemon is created for the job. Once requested, local garbage collection is invisible to the application.

local object
A system object that is local to a particular job. When a job terminates, all its local objects are deallocated.

local SRO
An SRO used to allocate local objects. Each job has one local SRO.

local variable
Local variables exist only for the duration of a job. A variable created or modified by a program is local to the creating job, unless specified as global.

low-level scheduling (dispatching)
Assigning a process to a processor. Each processor has a pointer to a dispatching port. When a processor is available to execute a process, it dequeues the first process from the highest numbered, non-empty queue in the port, and executes it.

M

M

2^{20} = 1,048,576. For example, 1M bytes equals 1,048,576 bytes.

mandatory argument

An argument that must be entered as part of a complete command.

mass storage service

The OS service that manages disk and tape storage.

master AD

The first access descriptor stored in passive store for a particular object. An object's passive version is deleted when its master AD is deleted. If a master AD is stored in a directory entry and other directory entries on the same volume set reference the same object, then deleting the master AD converts the AD in one of those other entries to a master AD, preserving the object.

medium-level scheduling

The process of dynamically assigning priorities to executing processes. Medium-level scheduling considers a process's running priority, service class, and dynamic behavior.

memory type

The kind of memory used by a system object, either normal memory or frozen memory.

menu service

The OS service that manages menus.

message

(1) Information issued by an executing program in response to some internal or external incident. A message can have three levels (short, long, and help) and can exist in various message languages (English, German, etc.). **(2)** Information used in executing the action associated with an SMS event. For an action class of `command`, the message becomes a process global that contains information for the batch job that is triggered by the event. For an action class of `mail`, the message is sent to the mailboxes listed in the action refinement.

message file

The container for a program's messages.

message service

The OS service that manages system and application errors and messages.

message stack

A stack that can be used to push and pop messages as execution continues. A message stack can thus contain a traceback of an error's propagation path from the point of error back through the various layers of software to the topmost level. Each process has a message stack associated with it.

menu

A list of choices provided by a program. There are two types of menus: "pull-down menus" from Window Services, and "screen menus" from the Menu Facility. Pull-down menu titles are displayed in a line at the top of a window; selecting a pull-down menu title causes the menu itself to be displayed. A screen menu (with its *menu items*) is displayed in a window under program control. Screen menus may have hierarchies of menus and submenus.

Menu Editor
System utility used to interactively create and modify menus.

Menu Handler
Ada package that processes menus.

menu item
Element of a *menu* representing one of the choices available in the menu. Composed of the displayed `menu item text`, and the returned `menu item index`; see the `Window_Services` package.

mode
The mode of a variable is either "read-only", meaning that the variable can only be read, or "read-write", indicating that the variable may be read or assigned a value.

modify rights
One of three type rights. By convention, modify rights are required to change an object's state.

monitor service
The OS service that supports monitoring of program execution.

multiple activation model
An activation model that activates an object in any job or node. Compare with *single activation model.*

N

name
(1) A character string label for an object or a stored AD. (2) A program-defined label for a program entity, such as a type, variable, constant, exception, package, or subprogram.

name space
A name space is a list of directories to be searched by the BiiN™ OS when looking for an object. This is similar in function to the UNIX environment variable `PATH` or the MS-DOS `PATH` command.

named association
A BiiN™ Ada construct that binds a parameter or an aggregate member to a value; has the form `name => value`.

named notation
(1) Entering an argument value to a command by specfying the name of the argument. (2) A BiiN™ Ada construct.

naming service
The OS service that provides packages to manage pathnames, directories, and lists of directories.

node
A single BiiN™ hardware system. Multiple nodes can be combined into a single *distributed system.*

node pathname
A pathname with one leading slash. The BiiN™ OS evaluates node pathnames beginning at the calling node's root (top) directory.

normal memory
Memory for system objects that can have pages swapped out to disk and that can be relocated by compaction. Contrast with *frozen memory*. Accessing a normal object may encounter delays waiting for pages or waiting for compaction to relocate the object.

null
(1) An invalid address, a pointer to nothing. (2) In general, empty or missing.

O

offset
An unsigned displacement from some base address, typically from the beginning of an object. An offset is in bytes unless other units are explicitly specified.

object
(1) A typed, protected memory segment. Such an object is also called a system object. (2) In Ada: a typed container for a value, such as a variable or constant. An Ada object may or may not be represented by a separate memory segment.

object address space
Up to 2^{26} system objects simultaneously addressable in a particular node's active memory.

object descriptor
A data structure used to hold various system object characteristics: size, location in memory, AD to the object's TDO, and other information. Object descriptors are internal to the OS; object descriptors are only described because it is difficult to explain how objects are located, sized, and typed without mentioning them.

object index
A field in an AD that identifies a particular object. In an active AD, the object index is a 26-bit index into the node's object table, selecting the object's descriptor.

object orientation
(1) A set of characteristics that enhance the coherence and security of integrated systems. The principal characteristic of object orientation is the use of protected data structures called objects to represent parts of the system itself as well as application entities. Objects are addressable and protected by cooperating hardware and software mechanisms. (2) An intuitive style of user interface that emphasizes representation of real-world entities rather than implementation-oriented details.

object representation
The contents of a system object. An object's representation can contain from zero to 4G bytes. The representation is not synonymous with the object itself because an object has several other characteristics, such as object type and attributes. Accessing an object's representation requires an AD or virtual address with rep rights.

object section
In the BiiN™ OMF, a contiguous portion of an object.

object service
The OS service that provides calls to manage objects, access to objects, and storage of objects.

object table
An object that contains all object descriptors for objects that are in a node's active memory or that have active ADs on the node. There is one object table per node. The object table is internal to the OS; it is described only because it is difficult to explain how objects are located, sized, and typed without mentioning the object table.

object tree
A collection of passive objects, beginning with a single root object, and linked by master ADs. An object x is in the tree if and only if x is the root object or another object in the tree contains x's master AD. Because master ADs cannot refer to objects on other volume sets, all objects in an object tree are on the same volume set as the root object.

object type
A set of object attributes that indicates such characteristics as its purpose, visibility, and usability by other system elements. Some types define objects that are recognized by the processor and for which special instructions are provided. Software-defined types can be manipulated only by a type manager corresponding to the type of the object.

object-specific attribute
An attribute that is defined differently or not defined at all on a per-object basis.

operator
A programming language element that specifies an operation to be performed on one or more operands in an expression.

operating system
The OS provides:

- General management of objects: object-oriented storage, protection, naming, and programming.
- Control and accounting for system resources, such as memory and processing recources, in a multiuser environment.
- Device-independent I/O access methods.
- Support for concurrent programming.
- Distributed services, so that applications built on those services are naturally distributed.
- High-level services commonly needed by many applications, such as messages, structured files, commands, forms, and reports.

System Services is the programmer's interface to the OS.

optional argument
An argument to a command that need be entered only if a value other than the default is desired.

organization pathname
A pathname with 2 leading slashes. The BiiN™ OS evaluates organization pathnames by first discovering which node to begin from, which may require a call to the Clearinghouse.

outside environment object (OEO)
An object that references the command definitions and messages associated with a program. These are used by the command language executive (CLEX).

P

package
An Ada module containing logically related types, constants, variables, exceptions, subprograms (calls), and tasks. A package is represented by two separate compilation units, a package specification and a package body.

package body
The implementation of an Ada package specification. The body includes implementations for each subprogram in the package specification, any private data and subprograms internal to the body, and any needed package initialization code.

package specification
The external interface to an Ada package. Declarations in the public part of a package specification can be used from outside the package. A package specification can also contain a private part that provides information needed by the compiler but not available to external users.

package type
A package specification that can have alternate bodies, with a body selected for each call depending on the object type of the first actual parameter. Compare with *attribute call.*

page
(1) A 4K-byte memory block, aligned on a 4K-byte boundary. (2) A printed page.

page descriptor
A data structure that locates a particular memory page and that contains access rights and status information for the page.

page table
A table that locates the pages of a paged object. The table contains an array of page descriptors.

page table directory
A page table that locates the pages of a large page table that is itself paged.

paged object
A large object that is stored in multiple pages of physical memory. The object descriptor for a paged object references a page table that in turn references the pages of the object.

paging
The process of moving pages between physical memory and a swapping volume set. Pages are loaded into physical memory on demand. Modified pages are written to the swapping volume set by an asynchronous paging daemon.

panning
Moving a view up or down in its frame buffer in order to see a different part of the frame buffer. Also called *scrolling.*

parameter
A value or variable that can be different for each invocation of a subprogram, and thus is supplied for each invocation. A formal parameter represents a parameter within a subprogram body. An actual parameter is the actual value or variable supplied for a particular invocation.

parameter mode
For an Ada parameter, one of:

`in`	The parameter is a value that is read but not written.
`out`	The parameter is a variable that is assigned but not read.
`in out`	The parameter is a variable that can be read or assigned.

passivate
Copy an active version of a system object to its passive version.

passive AD
An AD in passive store.

passive object
A system object in passive store, a passive version.

passive version
An object's version in passive store. An object can also have zero or more active versions.

passive store
The distributed object filing system for storing system objects on disk. Compare with *active memory.*

pathname
(1) A string of names that contains slashes and is a "path" of directories from a point in a directory structure to an entry. BiiN™ uses four kinds of pathnames: relative, node, organization, and full. (2) A series of base names, separated by slashes, that uniquely identifies an element in a form.

physical address
A 32-bit address of a physical memory location or memory-mapped device register.

physical address space
The 2^{32} byte address space used by the BiiN™ hardware.

physical memory
A node's semiconductor memory, whether normal RAM (volatile, read-write), battery-backed-up RAM (non-volatile, read-write) or ROM/EPROM/EEPROM (non-volatile, read-only for normal uses). Compare with *active memory.*

physical terminal
A video display device with a keyboard. It may also have a pointing device (mouse).

pipe
A software-defined object that supports interprocess communication (in one direction only). One process writes to the pipe and the other reads from it. The pipe uses a fixed-size buffer to hold data written by the first process but not yet read by the second process. The writing process will block if the buffer is full, and the reading process will block if the buffer is empty (the processes resume when these conditions no longer hold).

pointer
(1) A variable that contains the address of another variable or of a function. (2) In BiiN™ CL, an argument or variable type. A pointer value is a pathname to a passivated object.

port
An interprocess communications mechanism consisting of queued data structures that use shared memory and provide communications for processes within a single job. Ports contain messages, blocked processes, or are empty. Ports are the appropriate message mechanism when fast and simple message passing is needed.

positional notation
Providing the value of a command argument by specifying the value at the appropriate position in the command's argument list.

pragma
A directive to the Ada compiler, embedded in an Ada source file. Pragmas can provide important semantic information, such as how pointers are represented, or whether a subprogram can be called from another language.

print device
A device created by an application through which data is spooled or printed directly.

print service
The OS service that manages printers.

printer
An object that represents a physical printer connected to the system.

printinfo
A set of attributes describing the capabilities of a printer.

procedure
(1) A program unit in BiiN™ Ada, FORTRAN, or Pascal that is invoked by a call statement. Unlike a function, a procedure does not return a value. (2) In BiiN™ COBOL, a paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division. (3) In BiiN™ SQL, a collection of one or more SQL statements that can be called by a host language module. Procedures are grouped into SQL modules. (4) A program in CP microcode that forms a part of an IOM microcode program.

process
The smallest unit of scheduling; a single thread of execution; represented by a processor-recognized object. Processes specify execution environments for running programs.

process globals
A data structure that defines the environment in which a process executes. It is a list of ADs associated with the process.

process preemption
Forcing a running process to relinquish the processor to another process waiting in the dispatching port. It occurs if the waiting process has a higher priority than the running process and is a preemptive process (has a priority higher than the preemptive threshold).

processor claim
The number of time slices available to the processes in a job during each scheduling cycle. When the claim is exhausted, the scheduler terminates the job if it has exceeded its time limit, or obtains more processor claim if it hasn't (allowing the job to continue).

program
(1) A complete collection of software modules that are designed to accomplish a given piece of work. There are several kinds of programs: dialogue programs (which accept runtime commands), start-and-go-programs (which accept runtime commands), application programs, and system utilities. A program may be invoked interactively from the keyboard or batched in a BiiN™ CL script. An executable program is the linked version of a program. (2) In Ada, a program is composed of a number of compilation units, one of which is a subprogram called the main program. Execution of the program consists of execution of the main program, which may invoke subprograms declared in the other compilation units of the program.

program building service
The OS service that provides support for building programs: creation, execution, and debugging.

program object
The root of a network of objects that comprise a program. A program object is created by the linker and referenced by a program AD. The linker stores the program AD in a directory after creating the program. A program consists of a program object, a global debug table (GDT), an outside environment object (OEO), and one or more domain objects.

Program Services
The OS service area that provides support for concurrent programming, program building, and resource control.

protection service
The OS service that provides packages to manage users, IDs and authority lists.

protection set
List of IDs and associated access rights. A protection set is associated with an ID, and a caller must hold an ID that matches one in the protection set, with the appropriate rights, before the caller can access that ID.

public data object
An object containing data that can be referenced from other domains (domains that have an AD to the public data object in their static data objects.)

pull-down menu
A menu that is activated by a mouse and which appears only on explicit request of the user. After a user has selected *menu items* from the menu, the program can determine the menu choices by calling the appropriate *terminal access method.*

R

range
In BiiN™ CL, an argument or variable type. Range values are composed of two integers that are separated by a double period (`lower_integer..upper_integer`).

rank
(1) Default order in which spool files will print. (2) Default order in which subform group instances will be displayed in a form.

read rights
A type right required for many devices and opened devices, in order to read data using an I/O access method. Read rights rename use rights.

read rep rights
Rights bit that must be 1 to read an object's representation. ADs and virtual addresses contain read rep rights.

real-time job
A job that is executed in real time because it cannot wait for objects to be brought into memory or for another job to finish with a processor before executing. Real-time jobs have very high priority and infinite processor claim. They run in frozen memory, and are not subject to the scheduling process. If they block for I/O, the hardware reschedules them immediately.

real type
A simple data type that represents a floating-point number.

record
(1) In the BiiN™ OS, an element of a structured file. Each record in a structured file has a unique *record ID* that can be used to access the record. A record has a *format* that is either *fixed-length* or *variable-length.* (2) In COBOL, the most inclusive data item. The level-number for a record is `01`. A record may be either an elementary item or a group item. (3) In BiiN™ Pascal, a predefined type. (4) The unit of information in an object module. The BiiN™ Systems Object Module Format specifies about a dozen records, each of which contains specific information about the object module. These records are a header record, various symbol and object definition and reference records, and an end-of-module record.

record access method
An access method that transfers data in record-like units, in various access modes.

record type
A structured data type consisting of a fixed number of components (fields), possibly of different types, that are referenced by means of identifiers.

recovery agent
Process provided on each node by the OS that detects I/O processor failures and maintains a

table of existing I/O messages. Device managers keep this list current by calling `DD_Support.Register_IO_message` each time they create an I/O message.

region

(1) An area within a form. Valid regions are: the form as a whole, a subform, a group, a screen field or an enumeration. (2) A linear address space is partitioned into four 1-gigabyte system objects called regions. Region 0 contains static data, region 1 contains instructions, region 2 contains the stack, and region 3 is used by the operating system. Calling another domain in the current subsystem can change regions 0 and 1. Calling a domain in another subsystem can also change region 2. If a region contains less than one gigabyte, then the linear address space contains invalid parts. Reading or writing with an invalid linear address raises `System_Exceptions.length_violation`.

relative file

A structured file whose records are organized in an array of fixed-size record slots that may or may not contain information. A relative file can be read or written in any order.

relative pathname

A pathname with no leading slashes. The BiiN™ OS evaluates relative pathnames relative to a specific directory; by default, the current directory.

rep rights

Rights bits required to read or write an object's representation. ADs and virtual addresses contain rep rights. There are two rep rights: read rep rights and write rep rights.

representation type

An object characteristic that specifies which of the four kinds of object representation is used: embedded, simple, simply-paged, or bi-paged.

report

A printed or displayed document containing labelled data, often presented in columns and hierarchical groups with subtotals and totals.

report description

A DDef that describes the format of a report and the data to be printed in it.

report service

The OS service that manages reports.

reservation service

The OS service that supports the reservation of devices for exclusive use by a session.

resource priority

A process's resource priority. When an interactive or batch process requests the use of a resource (for example, a disk), the process's priority is raised to the sum of its base, bias, and resource priorities (but still in the range 1 to 10).

resource service

The OS service that supports resource control and accounting.

rights

Bits in an AD that control access to a system object. There are two kinds of rights: rep rights and type rights. Rep rights are required to read or write an object's representation. Rep rights are checked and enforced by the CPU. Type rights are required to invoke certain type manager calls with an object. The interpretation of type rights varies for different object types. Type rights are checked and enforced by type managers. Rights are not interpreted in null ADs.

rights mask

A record representing rights to be checked, added, or removed in an AD.

running priority

The priority at which an interactive or batch process is currently running. It fluctuates between the process's base priority and the priority of the resource the process requested most recently.

runtime command

A command that is processed by a program, using the command service. Runtime commands are defined in command sets. Command sets can be stored in the program's outside environment object (OEO), or as separate objects.

S

scalar type

A data type whose variables have a single value; also called a *simple type*.

scheduler

A collection of hardware and software entities that together schedule the execution of jobs (and thus processes). The scheduler seeks to maximize the use of system resources by scheduling processors, physical memory, and I/O devices.

scheduling service object (SSO)

An object that determines the type of scheduling a job receives by specifying the job's service class, priority, time slice, memory type, initial age, and age factor. An SSO is associated with a job when the job is invoked. The system administrator is responsible for creating different types of SSOs and controlling access to them, thus controlling the type of service granted to different jobs.

scheduling service

The OS service that manages scheduling of jobs and processes.

scope

(1)The part of a form in which an element exists and can be referenced. A form element is in a form, or contained in a subform, a group, or a pile, i.e., in another form element. At any one time the editing scope extends only to elements located directly in the form, or directly in a subform or group, or directly on a pile. Only elements in the editing scope can be edited. (2)The portion of a program in which a program entity exists and can be referenced.

scrolling

Moving a view up or down in its frame buffer in order to see a different part of the frame buffer. Also called *panning.*

semaphore
An object for controlling and synchronizing access to data that may be shared by concurrent processes.

sequential file
A structured file whose records are organized in the sequence they are physically written. A sequential file must be read in exactly the same order that it was written.

service
A logically related set of packages or other program modules. A service provides completely procedural solutions to problems. Applications call services on behalf of users, but users do not directly interact with services. Compare with *tool* and *utility*.

service class
Denotes the general class of service a job is to receive. Four service classes are defined: realtime, time-critical, interactive, and batch.

service area
A logically related set of services.

session
A grouping of jobs belonging to one instance of a user's interaction with the system. A session typically contains several jobs. A session is usually an interactive logon/logoff period, but can also be the running of a batch command file.

set
In BiiN™ Pascal, a predefined type.

simple object
An object representation that fits entirely into all or part of one memory page. A simple object's size ranges from 64 bytes to 4K bytes.

simply-paged object
An object representation that requires multiple memory pages, but with a page table that fits entirely into all or part of a memory page. Compare with *bi-paged object.* A simply-paged object's size ranges from 8K bytes to 4M bytes.

single-activation model
An activation model that activates an object only in a particular home job (for local objects) or home node (for global objects); another job or node that attempts to activate the object instead activates a *homomorph*, a token object that stands in place of the actual object.

spin lock
A synchronization device used during the processing of I/O messages with calls that raise and restore interrupt handler priority levels.

spool file
A buffer maintained by a spool queue that holds data from print device objects which is to be printed.

spool queue
A spool device that must be installed before anything can be printed.

spool service
The OS service that manages spoolers.

SSO priority
The priority defined in a job's SSO.

stable store
Non-volatile RAM storage that is used to optimize I/O throughput from active memory to disk. Using stable store, writes to disk can be delayed indefinitely, which greatly reduces I/O access time.

stack
System object that provides a stack of frames that each contain the state of a particular subprogram call.

standard kernel image
Factory-supplied OS preconfigured to run on a system disk and a console terminal.

starter image
A self-contained, linked image that does not need a secondary store (such as a disk) for operation, and which is booted into memory from a distribution channel (such as a tape) for the sole purpose of executing certain system utilities to prepare the physical system to be operable under an OS standard kernel.

statement
(1) A program construct that defines actions to be performed by the program. (2) A source program construct at which a breakpoint can be set when using the BiiN™ Application Debugger. In general, any construct that is considered a statement in the formal definition of the language is also considered a statement by the debugger. However, the following constructs are *not* considered statements for debugging purposes:

- Any declaration in any language (or definition in C) other than a variable declaration (definition) involving dynamic initialization or a subprogram declaration (definition).
- Any declaration (as opposed to definition) in C.

In addition, subprogram declarations are *always* considered statements by the debugger, regardless of their treatment by the source language.

static data object
System object that contains the data for a particular domain. This object represents the static data region (region 0).

storage resource object (SRO)
An object used to allocate other objects. An SRO provides access to available memory and to available object table entries. The SRO used to allocate an object determines the object's memory type and whether the object is local or global. Each job has a local SRO, used to allocate objects local to the job. Each node has two global SROs, one for normal memory and one for frozen memory.

stream file
A stream of bytes that allows random byte positioning. This UNIX-like file organization is useful if you simply want to read and write bytes.

string
(1) In BiiN™ CL, an argument or variable type. String values are sequences of characters, enclosed in single or double quotation marks (e.g., `'string'` or `"string"`). If there are no spaces, tabs, or linefeeds in a string, the quotation marks are optional. One string subtype is *enumerated*, for which a set of allowable string values is defined. (2) In standard Pascal, a sequence of one or more characters, enclosed by apostrophes, representing a value of type `CHAR` (if a single character) or of type `PACKED ARRAY [1..`*n*`] OF CHAR`, where *n* is a positive integer equal to the number of array elements. (3) In BiiN™ Pascal, `STRING` is a reserved word, used as a type denoter.

string list
In BiiN™ CL, an argument or variable type. String list values are sets of strings, enclosed in parentheses (e.g., `(string1, string2, string3)`). The string values may be separated by spaces, tabs, or commas. If a string list contains just one string value, the parentheses are optional.

structured file
A file containing records of either fixed or variable length. Structured files optionally can have indexes. Structured files are useful if you need a way to maintain record structures. Structured file I/O is typically accomplished using record I/O. A structured file can have one of these organizations: *clustered*, *hashed*, *relative*, *sequential*, or *unordered*.

subnet
Informal term for subnetwork.

subnet service
The OS service that provides network-independent communication between nodes within a subnet.

subprogram
(1) A procedure, function, or subroutine written in any BiiN™ programming language. (2) In a form, a processing routine or key catcher.

subprogram type
An Ada subprogram specification that can have alternate bodies.

subtransaction
A transaction that is contained within another transaction.

subsystem
One or more domains that share a common stack (that is, they have a single subsystem ID).

Support Services
The OS service area that provides common definitions and utility packages that are of use to all other services.

swapping volume set
A volume set providing external storage for virtual memory.

symbolic link
A symbolic link contains a pathname. Symbolic link evaluation retrieves whatever AD is stored with that pathname.

System Configuration Object (SCO)
A sequence of configuration commands that attach and start configurable objects during the booting of the system to put the configurable objects into operable states.

system SCO
A sequence of configuration commands that attach and start those configurable objects (typically hardware components) required to complete node initialization of the OS.

T

tag bit
A 33rd bit that tags each memory word and indicates whether the word contains a valid AD. A tag bit of 1 indicates a valid AD. A tag bit of 0 indicates a data word or a null AD.

tail object
An object that must be *attached* to a configurable object before it can become functional.

temporary file
A file that is unnamed when created and exists only for the duration of the current job (unless explicitly named and saved).

terminal access method
One of two currently supported methods for procedural interaction with a terminal: character (`Character_Display_AM`), or graphics. Contains calls to access the screen and input devices.

terminal service
The OS service that manages terminals and windows.

time-critical job
A job that has less stringent time constraints than a realtime job. Time-critical jobs have the same priority as realtime jobs, but limited processor claim (they are rescheduled in round-robin fashion when a time slice expires). They need not run in frozen memory, since their time constraints can tolerate page faults.

time limit
The total processing time available to a job (and its descendant jobs). When the processes in a job exhaust the job's processor claim, the scheduler terminates the job if it has exceeded its time limit, or obtains more processor claim if it hasn't (allowing the job to continue).

time slice
The amount of processing time assigned to each process in a job in each dispatching cycle. (It does not include time spent on interrupts, processor preemption, or waiting at a port or on a semaphore). When a process exhausts its time slice, it is generally redispatched with the same time slice value. However, each job has a processor claim value that determines the

total processor time available to all the processes in the job. When the job's processes have used *n* time slices and exhausted the processor claim, the job is reexamined by the scheduler and either terminated or granted additional processor claim (and the processes resume execution).

timing service
The OS service that manages system time, timed requests, time computations, and time format conversions.

TM concurrent programming service
The OS service that provides concurrent programming support for advanced type managers.

TM object service
The OS service that provides object and memory operations for building advanced type managers.

TM transaction service
The OS service that manages transactions within a type manager.

transaction
A system object that groups related operations so that either all the operations succeed, or all are aborted and undone.

transaction service
The OS service that provides calls to start and resolve transactions.

transaction stack
A per-process stack of transactions. The top transaction on the stack is the *default transaction* for any transaction-oriented operations.

transport service
The OS service that provides network-independent communication between nodes.

type
A label that distinguishes one kind of entity from another. The type of an entity typically determines the entity's allowed values, allowed operations, and representation.

type definition object (TDO)
An object that represents one type of system object. A TDO contains type-specific attribute entries for the type. These attribute entries are inherited by all objects of the type.

type manager
A program module that conceals the representation of an object type and that provides all basic operations for the object type. One module may act as a type manager for more than one object type. Several type managers that work closely together to manage some aspect of the system (for example, filing) constitute a "service".

type rights
Rights bits required to invoke certain type manager calls with an object. ADs and virtual addresses contain type rights. There are three type rights: use rights, modify rights, and control rights. The interpretation of type rights varies for different object types. A type manager may also rename the type rights that it uses.

type-specific attribute
An attribute that can only be defined once for an object type. The attribute entry is stored in the object type's TDO. All objects of the type inherit the attribute entry.

Type Manager Services
The OS service area that provides packages to build *type managers*, software modules that implement new object types and their attributes.

U

unique identifier (UID)
An identification number that is never changed or reused once it is assigned to a particular entity. A UID securely identifies the entity for all time and all systems. For example, each BiiN™ node is assigned a UID.

unbounded global object A system object that is not local to any job and that has an unbounded lifetime. An unbounded global object can be reclaimed by global garbage collection when it is no longer accessible via any AD.

unordered file
A *structured file* whose records are organized according to available free space.

use rights
One of three type rights. By convention, use rights are required to read an object's state.

user
(1) In general, one entity using the services of another. For example, a program is a user of system services. (2) The person sitting at the terminal issuing commands and entering data.

user interface
The part of a program that accepts user input, displays messages, and creates output.

user SCO
A sequence of configuration commands that attach and start configurable objects (typically software modules) of a configuration that are not required to complete node initialization of the OS.

utility
Program or BiiN™ CL script that is invoked interactively from the CLEX > prompt. It is supplied by the system to perform a particular service for some group of users. Developers may create new utilities. A utility may or may not have *runtime commands*.

utility service
The OS service that provides system definitions, texts, string lists, and long integers.

V

variable

(1) A datum whose value can change during program execution. (2) In CLEX, a named and typed datum containing a value; also called an *environment variable*. A variable's *mode* is either "read-only" or "read-write". A variable's type is one of: *boolean, integer, pointer, range, string*, or *string list*. A variable may be read (and, if "read-write", set) either interactively (using the *built-in commands* for variables: `create.variable`, `list.variable`, `remove.variable`, `set.variable`) or procedurally (using the *environment service*). The scope of a variable may be either *global* or *local*. Passivated variables are stored in *variable groups*; some groups are predefined for use by CLEX, programs, and scripts. Variables are stored and passivated with the `manage.variable_groups` utility. (3) In FORTRAN, the term "variable" does not include array elements. (4) In COBOL, a data item whose value may be changed by execution of the object progam. A variable used in an arithmetic expression must be a numeric elementary item.

variable group

A group of BiiN™ CL (environment) variables, associated with one or more BiiN™ services, programs, or applications. A variable in a variable group is identified by the group name, a period, and the variable's name. For example, CLEX uses the `cli.` (command line interface) variable group, which contains the current directory's pathname, command input prompt string, and so on.

version

(1) In general, a variation of a file that reflects the state of its development. (2) In the BiiN™ Software Management System, a member of a version group. A version captures a point in the evolution of a file (object).

view

(1) In BiiN™ SQL, a view is a named query that may be used as a table. In effect, views are virtual tables derived from the underlying base tables. They do not take up physical space. (2) A copy of an image module that makes available only a subset of the procedures defined by the image module from which it was derived. Executable programs may be linked to views, much like image modules and linker libraries. Views are a form of information hiding. (3) The visible part of a frame buffer.

virtual address

A location within an object, given by a 32-bit byte offset and an AD to the object. A virtual address can also be null, referencing no object. An active virtual address contains two words aligned on a word boundry. The first word is the offset; the second word is the AD.

virtual address space

Up to 2^{58} bytes simultaneously accessible: Up to 2^{32} bytes in each of up to 2^{26} system objects.

virtual memory

A memory management feature that supports a logical view of memory (for example as a collection of varying-size objects) that is distinct from the physical address space. Virtual memory requires hardware address translation, which is provided by the CPU. Virtual memory also implies support for logical memories larger than the physical memory, with the obvious problems being avoided by juggling parts of memory to and from disk.

virtual terminal
A device which, to an application, appears indistinguishable from a physical terminal. It provides a screen-like drawing space for the output of characters or graphics, and a keyboard and mouse for input.

volume
Logical storage area for storing files and objects. Volumes are members of volume sets.

volume number
A sequential number assigned to each volume in a volume set when created that identifies it relative to other volumes on the volume set.

volume set
A logical disk containing volumes used to store files and objects. Volumes of volume sets can span multiple physical disk devices.

volume set name
Name assigned when a volume set is created. It must be unique on all disk volumes that contain the volume set's volumes.

W

window
A portion of a terminal screen in which I/O can occur.

word
A unit of memory containing 32 value bits and an associated tag bit. A word is always aligned on a 4-byte boundary. Value bits in a word are numbered from 0 to 31.

work queue mechanism
A work queue data structure and two associated interrupt handlers designed to aid device driver writers in maintaining and initiating I/O requests for directly-connected devices.

working set model
A model for the reclamation of primary memory pages. The working set of a job is dynamically defined as the set of primary memory pages referenced by the job in the last time quantum, *T*, measuring backwards from a given time *t*. Every *T* time units the scheduler determines the working set for each running job. Any pages that have not been accessed in that time period are returned to a pool of free pages.

write rights
A type right required for many devices and opened devices, in order to write or change data using an I/O access method. Write rights rename modify rights.

write rep rights
Rights bit that must be 1 to write an object's representation. ADs and virtual addresses contain write rep rights.

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

We'd like your comments . . .

Your comments help us produce better documentation. Please fill out the form on the reverse of this card, then fold it, tape it shut, and drop it in the mail. (Postage is free if mailed within the United States.) We will carefully review your comments. All comments and suggestions become the property of BiiN™.

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 226 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

BiiN™
Technical Publications Department
2111 NE 25th Ave.
Hillsboro, OR 97124-9975

Please use this form to help us evaluate this manual and improve the quality of future versions. Mailing instructions are on the other side of this form.

If you want to order publications, see Page ii of this manual.

Manual Title: ______________________________ Revision Number (Page ii): __________

Please fill in the squares below with a rating of 1 to 10, with 1 being worst and 10 being best:

[ ] Technical completeness
[ ] Technical accuracy
[ ] Readability
[ ] Organization
[ ] Usefulness of material for your needs
[ ] Quality and relevance of examples
[ ] Quality and relevance of figures

If you gave a 4 or lower in any category, please explain here:

Please list any suggestions you have for improving this manual:

Please list any technical errors you found:

Please tell us your name and address. (If you would like us to call you for more specific comments about this book, please list your phone number as well.)

Name: ______________________________

Address: ______________________________

Phone (Optional): ______________________________

Thank you for taking the time to fill out this form.