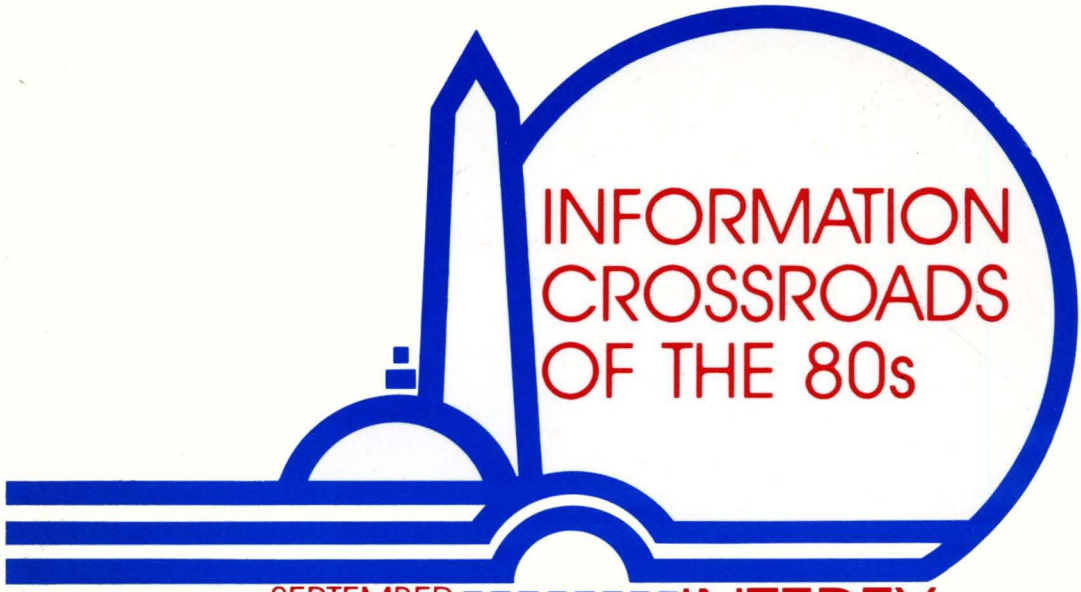


INFORMATION CROSSROADS OF THE 80s

SEPTEMBER 8 - 13, 1985

INTEREX

HOSTED BY BALTIMORE/WASHINGTON RUG

PROCEEDINGS

HP 1000/9000

INTEREX

the International Association of
Hewlett-Packard Computer Users

Proceedings

of the

1985 CONFERENCE

at

Washington, D.C.

Hosted by the
Baltimore-Washington Regional
Users Group

Papers for the

HP1000
and
HP9000

F. Stephen Gauss, Editor

Washington Conference Host Committee

Chris Sieger, Chairman
Tom Benedict
Nick Demos
Dan Felman
Steve Gauss
Sam Inks
J. J. Myatt
Don Nayden
Suzanne Perez
Joan Peters
Doug Pilkington
Frances Smithson
Holly Siler

Paper Review Committee

Cimarron Boozer
John Campbell
Dean Clamons
Dale Garcia
Art Gentry
Ken Griffin
Hugh Hanks, Jr.
Mark Katz
Jock McFarlane
Dick Minor
Terry O'Neal
Larry Rosenblum
George Santee
Dan Steiger
Szabolcs Szekeres
Steven Telford
Don Wright

Introduction

This volume of the Proceedings of the INTEREX 1985 North American Conference was printed from machine readable text supplied by the authors (with a few exceptions). Each paper was checked using the spelling checker SPELR (from the CSL library tape Rev. 2533) and printed using the version of TYPO (also from the CSL library) called TYPER on an HP2686 LaserJet printer. Papers have been numbered sequentially in order of presentation at the conference with HP1000 papers numbered 10xx and HP9000 papers numbered 90xx. Several papers will be of interest to members of both communities and it is quite likely that papers in the companion volume for the HP3000 and Series 100 will also be of interest.

Participation by the users of the HP9000 conference is significant (making up one third of the technical papers).

Thanks go to the authors who sent their papers in on time and to the paper review committee for their prompt response and illuminating comments. Through the efforts of Pam Tower we have an unprecedented number of papers from HP employees. Thanks also to Dean Clamons, Dan Felman, Mike Gauss, Karen Gauss, and Vivian Gauss who lent technical assistance at crucial times.

F. Stephen Gauss
U. S. Naval Observatory
1 August 1985

Index By Authors

* Papers marked with an asterisk (*) were not received in time for inclusion in the proceedings. The abstract is printed in its place.

Index By Title

* Papers marked with an asterisk (*) were not received in time for inclusion in the proceedings. The abstract is printed in its place.

1001. TEXED- TEXT PROCESSING PROGRAM FOR HP1000 COMPUTERS

Bill Hassell
Hewlett Packard Co.
3003 Scott Blvd.
Santa Clara CA 95050

ABSTRACT

TEXED is a program based on many of the text processing techniques currently popular today. It runs on RTE-4B, 6, and A opsystems and provides the user with a fast, well-featured documentation system. TEXED can provide a Table of Contents, an index, Appendices, and even has options to provide output that can be directed to the 2680 and 2688 laser printers. Justification, centering, change bars, and variable paragraphing can be invoked along with a page width up to 130 characters. A quick overview of text processing and TEXED's history will be provided along with ideas on how to create manuals, memos, and good-looking communications in general. TEXED was given to the CSL library in 1984 and an updated version that supports the new file system on both RTE-6 and RTE-A.

1002. THE ADA(tm) PROGRAMMING LANGUAGE ON HP HARDWARE

Arnold W. Kwong
DCS Corporation
363 Cretin Avenue South
St. Paul, Minnesota 55105

Larry Drews
Test Systems Division
Lexico Enterprises, Inc.
10516 Northeast 37th Circle
Kirkland, WA 98033

I. Introduction

This section will first describe the background of the ADA language and then discuss why ADA is important to the HP user community.

A. What is ADA?

The ADA programming language was developed under the auspices of the US Department of Defense. The project that led to the development of the ADA programming language was begun in 1975.

The needs that led to the development of ADA were:

1) A proliferation of languages (JOVIAL, CMS-2, various assembly languages, and lesser known languages such as TACPOL) were being used to develop software for systems used by the DoD.

2) The languages in use by the DoD, and its contractors, were often of poor quality which led to the failure of DoD development projects due to language and compiler problems.

3) The life cycle costs of designing, developing, maintaining and creating systems in specialized languages were growing.

ADA was specifically designed incorporating the experience gained in research and industrial environments using languages such as PASCAL, MESA, MODULA (MODCAL), ALGOL (SPL), and PL/1. Specific additions for the real time environments include inline machine code insertion, interfacing to other languages (including assembler), untyped variable data conversions, and time-to-interrupt task specification.

An important element in the development of ADA as a programming language is the concurrent development of a "programming environment" within which ADA language based applications are developed, executed, and maintained. Generically these environments are called "APSE"s ("ADA Programming Support Environments") and are labeled in three types of special interest: the "KAPSE" ("Kernal ADA Program Support Environment"), the "MAPSE" ("Minimal ADA Program Support Environment"), and the "APSE" ("ADA Program Support Environment").

The KAPSE includes operating system functions such as resource allocation, input/output, process management, and error handling.

The MAPSE includes basic program development tools such as an ADA compiler, linker, absolute object loader, debugging tools, editor, prettyprinter, terminal interface, file system, software configurator, and command interpreter.

The APSE includes environmental tools not usually associated with a specific programming language. These tools include: context/syntax directed editors, documentation support, project control, software configuration control, performance measurement, error tracking, design and verification, pre/post processors, command interpreters, and support program code libraries.

In summary, the ADA programming language environment consists of more than a language compiler. The environment includes defined levels of interfaces, tools, and methodologies that combine to provide an effective means with which to develop application systems.

B. Why is ADA important to HP users?

The original broad design for ADA was focused by DoD Directive 5000.31 that mandated the use of ADA for "mission critical" systems begun in 1984. The definition of "mission critical" encompassed such embedded computer uses as avionics, fire control, and aircraft control. This definition has since been extended in practice to include a wide variety of other systems.

The HP user sites in the DoD and its contractors thus have very strong reasons for being interested in ADA. Other HP users need to consider the advantages that ADA will bring in the long term:

1) Ease of maintenance, leading to lower life cycle costs, is a critical design goal of ADA.

2) Efficiency was an almost overriding concern during the language development process.

3) ADA presents a machine independent consistent method for describing and using parallel and sequenced tasks. The reduced need to train programmers in the idiosyncratic use of a specific operating system is a benefit in building and maintaining systems.

4) ADA compilers have a better chance of working properly after validation. A DoD validated ADA compiler has to pass a compile/execute suite of over 2600 tests. This gives a greater assurance that the compiler works in the documented manner.

5) ADA provides a well thought out method for handling faults, errors, and processing exceptions. This, and other features of the language, provide for the design and development of maintainable fault tolerant systems within real time operating environments.

6) ADA is specifically designed to facilitate the transport of tools between ADA based systems. The MAPSE/APSE level provides the means

for vendors, and third party software suppliers, to supply sophisticated tools for a very wide market without having to be concerned with the intricate details of each of the underlying operating systems.

7) Funding for development of software development methodologies, training materials, and software tools is being provided by the DoD. The availability of tools and trained people is a long term advantage.

II. How to get ADA on an HP machine: A quick tutorial on software architectures.

This section will discuss the needs of the ADA language for KAPSE and instruction set features, and then a general discussion of HP systems' architecture will be made. Following the discussion of the HP systems' architecture will be a short tutorial on virtual systems and approaches to implementing them.

A. The ADA KAPSE and Instruction Set Requirements

Implementing ADA environments (KAPSE/MAPSE/APSE) is a time consuming and difficult task. Existing efforts have exceeded two man centuries. Fortunately the effort required to move an existing compiler does not approach this magnitude.

The STONEMAN Document forsaw the use of virtual environments in order to implement ADA in a variety of ways: cross compiling, rehosted environments (developed and run on separate machines), and code generation for multiple machines from a single compiler design. (5.F.1) Included in the list of possible machine-independent low level coding schemes were: BCPL/C, P-code (PASCAL), JOVIAL, and others.

Implementing ADA requires implementing several "virtual systems" where the implementation of each layer does not depend on implementation details of other layers.

The KAPSE

At the next to lowest level, the Kernal ADA Program Support Environment (KAPSE), the interface to the operating environment, takes the form of calls to system services. In practice the current implementation of KAPSE functions has followed the provisions of the UNIX Kernal. The Telesoft implementation of the ROS KAPSE required only about 300 lines in the C programming language to implement the Kernal functions required for ADA as a virtual system above UNIX. This kernal approach is not unlike that of the HP 3000/MPE Kernal, HP 9000/500 Series SUN Kernal, or the services provided by the HP 9000/200 Series PASCAL environment.

The Instruction Set

At the lowest level the ADA compiler must create a list of instructions to be executed by the macro-level instruction set of the executing processor. There are several approaches to deciding what are the "best" instructions to execute for efficient program processing.

The ADA language is very demanding on the features of the instruction set that

is running the compiled code. There are many requirements for run time checking, complex fault handling, references to complex data structures, and rules for data sharing. This provides significant challenges in the current HP systems.

HP systems have historically been microprogrammed for specific language and feature support.

The HP 1000 traces its roots back to "The Gibson Mix" that was characterized at IBM. Later HP 1000s have extended the original instruction set with 'RPL' instructions that serve to implement a wide variety of support functions for applications (such as the Fast Fourier Transform), computation (such as the vector instructions), language support (such as the Fast Fortran and Decimal String Arithmatic), and architectural extension (such as Code and Data Separation). Thus, the HP 1000 is optimized for the efficient execution of FORTRAN and support for the RTE operating environment.

The HP 3000 traces its roots back to the Burroughs B5000/5500 systems and the use of the ALGOL-derived Systems Programming Language (SPL) for most of the system. The HP 3000 instruction set has been microcoded on at least four different architectures that completely emulate the commonly used user functions so as not to require recompilation of programs in all cases. The microcode for the HP 3000 also includes features that increase the efficiency of MPE operating system functions, COBOL language features, and FORTRAN language features.

The HP 9000/200 Series systems use the Motorola 68000 as a processor. The M68000 has been custom remasked for IBM to emulate the majority of the IBM 360/370 unpriviledged instruction set for use in the XT/370 and AT/370. The internal use of MODCAL to develop some of the underlying support for the UCSD-derived PASCAL environment is a demonstration of the multiple levels used in the 9000/200.

The HP 9000/500 Series systems have provisions for on chip-set microcoding. This is rumored to have been the basis for the canceled VISION product which would have run the HP 3000 instruction set. The existing use of both MODCAL and C demonstrate the current orientation of the instruction set microcoded on the HP 9000/500.

Other HP systems, such as the HP 64000 and HP 250, use descendents of the HP BPC chip set which was originally designed for use in the HP 9825 calculator. This cpu has a limited memory addressing space and a limited instruction set.

In summary, the demands of the ADA KAPSE and language features provide a difficult set of problems for would-be compiler and environment implementors. The optimization of the major product lines, the HP 1000 and HP 3000 systems, for other languages and environments introduce obstacles for a new compiler and environment like ADA.

B. Emulating Systems Using Virtual Systems and Virtual Instruction Sets

This section will define and describe the use of "emulation" in order to implement features on systems not necessarily optimized for a given language.

The use of emulation in order to run a system of different characteristics on a system normally used for other functions is not widely known, but the technique has been used in many commercial systems.

A "virtual instruction set" approach allows for the execution of an arbitrary instruction set on hardware and firmware that may be optimized for other uses. The "target" (or emulated) instruction set is executed by hardware and software such that the executing program cannot distinguish the execution environment from one of pure hardware designed specifically for it.

"Virtual systems" are implemented in virtual instruction sets to execute on hardware and firmware other than that specifically designed to execute the virtual instruction set.

"Virtual machine" operating systems allow for the execution of multiple operating system copies on the same physical hardware. Thus, concurrent execution of the IBM DOS and MVS systems is possible with the use of the IBM VM Operating System. The IBM VM operating system treats the other operating systems as "guests" which are running on the same physical hardware. Each of the guest operating systems may execute special instructions that are unknown in the other guest, or the VM, operating system.

The existing examples of virtual instruction set machines include the Burroughs B1700/1800/1900 Series, the Nanodata QM/1, and the APL Assist feature on the IBM 370 Series.

The effective use of virtual system techniques along with instruction set virtual machines provide an operating environment that cannot be distinguished from the emulated machine. In this paper the term "emulation" will encompass both the virtual system and instruction set virtual machine use to provide the apparent system image. Examples of these machines include the IBM 1400 Series emulation on IBM 370/30XX, emulation of the B2XX machines on the B1000s, and emulation of the APL/SV environment on the IBM 51XX.

Emulated Machines

The emulation of a target virtual system and instruction set can be performed totally in software, although this normally incurs a high performance penalty when contrasted with assembly language coding for the same task.

Examples of the software implementation of emulated virtual systems include the PASCAL-P/UCSD Pascal, many BASIC interpreters (including BASIC/3000), APL/APLGOL on the HP 3000, and RM/COBOL on many small systems.

Why use a software virtual machine for a spectrum of need?

Emulating a target machine in software is needed in several situations:

1) The emulated machine is obsolete, but some applications should not, or can not, be converted to a new system. Examples include the IBM 14XX Series emulated on the IBM 370/30XX, the Honeywell 2000 Series emulated on the Level 64/DPS 7, and the IBM 1130 emulated on the META 4.

2) The emulated machine does not yet exist. Emulation is frequently the preferred approach where the target machine is not yet available or fully debugged. Examples include the HP 9000/500, the HP 3000/44, the HP 300, and the HP 150.

3) The emulated machine will never by built as dedicated hardware, but the virtual system and virtual instruction set are efficient approaches to solving problems. Examples include the E-Code used in HP APL/APLGOL/3000, the P-code used in UCSD PASCAL (also implemented in hardware as the Western Digital Microengine), and the RM/COBOL intermediate code.

4) The emulated machine may need features that the existing hardware cannot support directly. Examples are the COBOL implementations found on small systems. Packed decimal arithmatic, picture clause editing, byte oriented memory access, and extended precision arithmatic instructions are not usually found in small systems.

5) The emulated machine may be implemented using a simplified instruction set in order to reduce hardware cost and simplify CPU design. This technique is at the heart of the Reduced Instruction Set Computer (RISC) projects like Spectrum. Examples currently being available are: the APL/SV models of the IBM 51XX (50 instructions in the native machine), the Cyber 8XX Series (supporting both NOS and NOS/VE environments), and the Pyramid 90X (with a large register window).

The use of emulated machines can gain benefits not available using just hardware:

1) Certain guarantees about the execution environment (such as interrupt handling, time synchronization, resource sharing mechanisms, and fault/error handling) can be made where the native hardware base changes. A good example is the implementation of the HP 21XX instruction set in the HP 1000/A-900. The word I/O to and from cards is emulated even though the physical I/O occurs via DMA on the backplane instead of handshaking. The current product architecture is very different from that of the original HP 2114.

2) Debugging the system, performance measurement, portability, and development are much easier. The target being emulated need not be run on a small system when first prototyped. Large scale systems, with richer toolsets and faster speeds, can be used until hardware becomes available. Emulation is a tool that can be used to guarantee the portability of source level codes although the underlying hardware may be altered significantly. Examples include HP Standard PASCAL which uses similar internal structures prior to code generation.

3) Different environments can be provided on the same hardware. Although the IBM 14XX emulation may be active, other tasks can be concurrently processed within the IBM 370. This concurrency of processing is due to the allowances for separate virtual systems

(and instruction sets) executing over the same physical equipment.

In some cases the development of virtual systems and virtual instruction sets has led to the creation of a special optimized CPU for the "virtual" environment. Examples include the BellMac 32000 (for O-Code/C), the Western Digital Microengine (for P-Code/PASCAL), and the INTEL iAPX-432 (for ADA).

C. Summary

Emulating a virtual system, and possibly a virtual instruction set, can be an effective means to implement systems, software, and features that cannot otherwise be supported on a given system. This technique can be used to implement ADA on the HP 1000, or other HP, systems.

III. Implementing ADA on HP Systems

This section will discuss the efforts to implement ADA on HP systems and then examine the suitability of existing HP systems for execution of ADA environments.

A. ADA Implementation Efforts on HP Systems

There are, or have been, at least three projects to implement ADA for HP systems.

The Solutions Plus ADA for the HP 9000/200 is a conversion of the Telesoft ADA compiler. This software is currently marketed and will reflect the validated Telesoft compiler as it becomes available.

There was an attempt to produce an ADA compiler for the HP 1000 by Science Applications, Inc. The effort was to generate a subset ADA compiler that generated HP 1000 instructions from ADA language programs possibly using the PASCAL code generators. This effort has been suspended.

A partial subset of ADA for the HP 3000 (approximately the size of PASCAL) was implemented by NuSoft, Inc. As far as is known this effort has also been suspended.

Hewlett-Packard has signed a contract with Alsys, a French firm headed by ADA designer Jean Ichbiah, to acquire the AlsyComp technology. Published information indicates that this compiler product consists of "... the machine independent root portion of the compiler, and the tools, utilities, and documentation needed for them to develop the remaining 20 percent (or back end) needed to target a compiler to a particular hardware...". Thus, the KAPSE/MAPSE/APSE development remains for HP. The initial validated version for the Alsys compiler was performed on an M68000 based system. HP plans for a product are not known as of this writing.

B. What Choices are there for ADA on HP Equipment?

This section will consider each of the HP product lines in turn and their suitability to execute ADA environments. ADA environments currently available,

or in development, will be noted in passing.

HP 3000

The HP 3000 computer systems would appear to be suitable for execution of ADA environments. The stack architecture and block-language (ALGOL/SPL) orientation provide some basis for ADA implementation. Significant drawbacks are present in the limited data stack size (64KB) and the overall processing power (approximately one MIPS). Existing ADA compilers consume DEC VAX 11/780 systems very quickly both from an I/O and CPU usage perspective. ADA compilers tend to be very large and require large linear address spaces for symbol tables and semantic checking of program code. The HP 3000 is not supported as a user microcodable system. The combination of limited data stack size, limited processing power, and no access to microcode support make implementing a full validated ADA environment on the native HP 3000 unlikely.

HP 9000/200 and HP IPC

The M68000 based HP 9000/200 is currently being supported by Solutions Plus with a conversion of the Telesoft ADA KAPSE and compiler. The large linear address space and large memory available makes the HP 9000/200 Series an attractive target machine for an ADA environment. A number of ADA environment/compiler producers are targeting the first implementation of their products to the M68000 based systems. (Alsys, Telesoft, others)

HP 250/260, HP 64000

The HP 250 and the HP 64000 share the use of the HP BPC chip set. It is unlikely that there will ever be an ADA environment converted for the HP BPC. The availability of PASCAL and C for cross compilation on the HP 64000 makes it possible to consider use of ADA environments for target microprocessors on the HP 64000. The ADA environment and its related tools would probably be implemented in a "target" processor. A question arises whether the HP 64000 would be the target system, or the system under development, when the majority of memory access and processing would be occuring outside of the HP 64000. The in circuit emulation and software performance measurement features of the HP 64000 would be ideal for ADA environments and some "cooperation" between an external ADA environment ("target") and the HP 64000 system would be useful. The implementation of this cooperation would be in the form of transferred symbol tables and a memory allocation map for use by the HP 64000.

HP 150

There are currently several efforts to develop an ADA environment for the INTEL 808X/80X8X microprocessors (INTEL, R&R, TELESOFT, others). A conversion of one of these products is possible, but no organization has announced plans to do so. The KAPSE would be layered on top of MS-DOS or XENIX for these CPUs. Another alternative is cross compilation support for the object code with an HP 150 resident KAPSE (INTEL).

HP 1000

The HP 1000 is used in environments where the ADA environmental capabilities

are most useful: real time control, data acquisition, test and measurement, and embedded processing. The existing HP 1000 instruction set architecture makes it difficult to implement a block structured language of significant complexity. The difficulties of HP PASCAL/1000 are evidence of problems mapping a block structured language to a FORTRAN oriented instruction set.

The HP 1000 can also use the M68000 add-in card (HP currently supplies one) and run cooperative ADA-generated code with the HP 1000 providing I/O and other services.

The availability of user microcoding on the HP 1000 provides the opportunity to extend the HP 1000 native instruction set to provide for the extended functionality needed by ADA.

C. Summary

The HP product lines have a number of opportunities for ADA environment implementations. The HP 1000 is the product line most suitable for ADA usage and one of the most difficult products on which to implement ADA.

IV. ADA on the HP 1000

A. Problems with implementing ADA on the HP 1000

There are a number of significant problems that face an implementor of an ADA language compiler for the HP1000. Chief among these are:

Multiple machines (E,F,A series)

Multiple operating/file systems

Archaic architectures

Non-tasking operating systems

No direct access to interrupts

Interfacing ADA programs to host operating/file system

Interfacing ADA programs to existing subroutine libraries

Each of these problems will be examined in turn.

1. Multiple machines (E,F,A series)

The HP1000 series of computers actually consists of a number of slightly different architectures. As each machine was introduced, additional features were added in response to customer needs. The A- series embodied several substantial enhancements.

In order to minimize development and maintenance costs, it is desirable to implement any ADA language system using a common set of capabilities that are available on all of the machines in the series. Unfortunately, some required capabilities fall outside of the common set and therefore cause multiple versions of the ADA language system to exist.

A prime example of a non-common feature which is different is the microcode architectures. As will be discussed later, one of the mechanisms that can be used to solve the ADA language implementation problems is the use of microcoding to provide necessary architectural features. The A900, A700, and the older E and F series all have different microcode architectures which will require different versions of any microcoded features.

2. Multiple operating/file systems

The HP1000 series of computers currently has several operating system/file managers combinations in use. On the E,F series, RTE-IVB and RTE6/VM are the primary operating systems in use. On the A series, RTE-A is the operating system, but there are two different file managers in use: FMGR and FMP. This gives four combinations of operating/file systems that the ADA language system must cope with.

3. Archaic architecture

The original HP1000 architecture is almost 20 years old! It started out life in the late 60's as the HP2116 (or was it the HP2114?). Computer architectures have evolved considerably since then. And while HP has made yeoman efforts to adapt the original design to accomodate the new requirements, still the HP1000 suffers from the lack of modern architectural features. Primary deficiencies include few registers, non-stack architecture, and limited address space. This results in a machine that cannot easily support re-entrancy or recursiveness, features that are inherent in the ADA language.

The A-series has implemented a limited stack for procedure calling and local variable storage, but such features do not exist on the E and F series. The limitations of the HP 1000 instruction set compatibility between series of machines is also a problem.

The saving grace of the HP1000 architecture is open access to its microcoding mechanisms. As will be discussed later, with microcoding many of the above architectural deficiencies can be overcome. In short, the implementation of a virtual system and virtual instruction set can provide an adequate cure for the archaic architecture of the HP 1000.

4. Non-tasking operating system

None of the operating systems on the HP1000 series provides for convenient user application multi-tasking. While separate programs may be invoked as asynchronous executing entities, storage sharing is severely restricted. Running multiple copies of a program using such a restricted set of capabilities would be difficult to implement. And yet, this is exactly the intended operating mode of ADA applications. Consequently, ADA multi-tasking becomes a problem area in any implementation on the HP1000.

5. No direct access to interrupts

The HP1000 operating systems do not provide user application programs any kind of direct access to interrupts. The user may implement special drivers, or even programs that are initiated by interrupts, but applications that must be able to recognize and handle multiple interrupt driven events have no mechanisms to handle such requirements.

A prime example is clock interrupts. Much of ADA tasking requires the ability to sense the passage of time on an interrupt basis. However, there are no mechanisms in the HP1000 operating systems that provide an application program with an interrupt signal to indicate that a given time interval has occurred or that a specified clock time has arrived. Getting around this deficiency represents a significant challenge.

6. File system interface

Although input/output operations are not part of the ADA language definitions, a standard package handling the major forms of file input/output is defined. On the HP1000, the ADA language system implementor must provide for an interface between this standard package and the the HP1000 file system. Given the number of file types available to the HP1000 user and their various characteristics, this file system interface is a non-trivial problem.

7. Existing subroutine library interface

There exists a large body of software already implemented on the HP1000 for various applications. For practical purposes, any useful ADA language system on the HP1000 must be able to access these existing subroutine libraries. The two major problems involved here are linking to the subroutines and passing of parameters. The ADA language does provide an optional mechanism to support this linkage, the "interface pragma." This, or some other pragma, must be implemented to provide the mechanisms necessary to allow use of the existing software.

B. Solutions to the problems

Given the problems outlined above, the burning question becomes: how can an ADA language system be implemented on the HP1000 series? As will be discussed, emulation with microcode assistance seems to be a workable approach.

1. Emulation of better architecture

Emulation allows the use of a virtual instruction set that supports the types of operations required by an ADA language system. Stack operations, re-entrancy, recursiveness, and multi-tasking primitives may be easily implemented given an appropriate virtual instruction set. Similarly, an efficient instruction repertoire in conjunction with easy and straight-forward code generation can compensate for some of the loss in execution time associated with emulation. Execution performance can be further enhanced by microcoding critical and frequently used portions of the emulator. In fact, with a suitably chosen virtual system implemented in a reasonably effective manner with microcoding where possible, the emulated programs will perform comparably to native HP1000 Fortran programs. This relative effectiveness of emulated instructions occurs

because of the inherent awkwardness of the native HP1000 instruction set in implementing high level language structures.

A major consideration in choosing such a virtual system is the availability of ADA compilers that generate code for such a virtual instruction set. If implementing a new code generator can be avoided, then the task of porting an ADA language system onto the HP1000 series will be eased considerably.

2. Lowest common denominator

Since the HP1000 series is a series of not quite identical machines, the lowest common denominator, in terms of machine features and operating system functions, must be found and used. To the extent that instructions or operating/file system functions that are not common across the series can be avoided, to that extent is the task of providing a uniform product eased.

3. Localize the differences

However, non-standard features cannot be totally avoided. The emulator must interface to the host I/O system, and will use the native microcode to improve performance. The goal in these areas is to localize the usage of these features so that the differences between versions of the ADA language system are well-defined and easily controlled. In addition, the A-series does provide enhancements in the architectural area that may well be worth using in spite of the additional maintenance problems that may be incurred.

4. Multi-tasking within one session/program

Since none of the HP1000 operating systems provide the level of sophistication in multi-tasking required by an ADA language system, those capabilities must be provided as part of the emulation system. This restricts multi-tasking to be contained within one session. While this is not the general, host operating system integrated true multi-tasking that is preferred, for most applications on the HP1000 this will probably be sufficient. However, it should be noted that this is a major deficiency that can not be rectified without introducing a new operating system.

5. Polling of central interrupt flag

Again, the host operating systems effectively prevent coupling ADA language applications with multiple interrupts, or with clock interrupts of any kind without significant surgery. A polling mechanism which is coupled to the emulation cycle can be implemented that can simulate interrupts somewhat crudely. Every so often, at whatever resolution is required, the emulator can poll the various sources of interrupts to determine if an interrupt has indeed occurred. If so, the emulator can fake whatever interrupt action is suitable for the emulated architecture.

6. Emulator implementation of low level file operations

If the emulator can recognize and trap the low level I/O operations associated with file manipulation, then the emulator can redirect the actions to the host operating system. The standard package defined for ADA systems uses file types

that are a subset of the FMGR and FMP file types, so no major problems should occur in this area. In essence, RTE becomes a virtual machine system where the ADA system is a microcode assisted virtual system.

7. Foreign subroutine interface

There exists a large collection of programs and subroutines already implemented on the HP1000, primarily in Fortran and Macro. These packages represent a substantial economic resource and will not be readily abandoned by users just because a new language has appeared on the scene. Consequently, any ADA language system that is implemented on the HP1000 must be able to access and use these existing software packages in a reasonable manner.

Since an emulator based system is being proposed, several problems arise in the area of interfacing to foreign subroutines. First, how to recognize that a foreign subroutine is being accessed, and secondly, how to pass parameters to and get results from the foreign subroutine. A special "pragma" must be introduced by the ADA language implementors. This pragma would associate a programmer specified number with the declaration of the foreign subroutine, and allow the emulator to trap and redirect the procedure call to the user provided subroutine via a user provided lookup table.

The virtual machine/virtual system approach allows each system to operate in an optimized manner (ala' the B1700 or QM/1).

C. Summary

Implementing an ADA language system on the HP1000 would be much simplified by using a virtual system suitable to the ADA language requirements. With micro-coded assistance, performance of such a system would approach native Fortran efficiencies. An emulation approach provides the controls necessary to provide multi-tasking and interrupt controls. And interfacing to FMGR or FMP files does not seem to be difficult. Finally, mechanisms to interface to existing subroutine libraries can be devised, which, although somewhat awkward, can do the job. ADA on the HP1000 awaits the putting together of these components.

V. Implementing MAPSE and APSE Capabilities on the HP 1000

This section will describe the possible MAPSE and APSE alternatives for use on the HP 1000.

A. MAPSE/APSE Features

The ADA MAPSE and APSE, as defined in the STONEMAN document, provide more sophisticated development support capabilities than are typically used on the HP 1000. Integrated tools that are specific to the ADA environment (pretty printers, cross references, software configuration control, and performance analysis) are not commonly available for the HP 1000. ADA environmental tools, such as the ALS being developed by SofTech, can be converted to a validated HP 1000 ADA environment with a minimum of effort. The effort is minimized by the use of the validated ADA compiler. KAPSE and MAPSE tools are generally coded in ADA thus providing an interchangable source of tools within the ADA community.

B. HP 1000 Considerations

In order to support sophisticated tools in the HP 1000 systems two problems have to be overcome:

1) Human interaction may need other methods than are currently supported by the HP 1000 (such as mouse, light pen, or voice entry).

2) Sophisticated tools put a load on the HP 1000 system where resources may not be available.

These problems can be overcome using intelligent workstations such as the IBM PC or HP 150 to off load human interaction and tool processing functions from the HP 1000. The existing tools for ADA on these workstations include context sensitive editors, syntax checkers, and methodology support for ADA as a program definition language.

The workstations can be combined with the computer linkage capabilities of PC-to-1000 transfer capabilities to effectively do the code creation on the workstations and then transfer the files to the HP 1000.

C. Summary

The tools available for the MAPSE and APSE levels will not be as complete as some larger machine environments. Sufficient tools are already available to make development of ADA software for the HP 1000 viable.

VI. Conclusions

The ADA programming language is a good fit for applications on the HP 1000. The implementation of a virtual system for ADA, using a virtual instruction set just for ADA, will provide adequate performance on an HP 1000. The implementation of ADA on the HP 1000 is feasible without a long period of time.

VII. Annotated Bibliography

INDEX TO BIBLIOGRAPHY

C. HP 9000
D. HP 300/3000

I. Support for Virtual Systems

A. B1700 References

Design of the Burroughs B1700; Wilner, W. T.; Proceedings of the 1972 FJCC; 1972; P 489-497

A definitive reference to the design of the B1700 systems this article describes the use of virtual instruction sets to allow an optimized implementation of various high level languages to be supported on a single hardware design.

Burroughs B1700 Memory Utilization; Wilner, W. T; Proceedings of the 1972 FJCC; P 579-586

This is a good reference comparing the use of optimized instruction sets tuned for a given source language (FORTRAN, COBOL, and RPG are the examples used).

Realizing A Virtual Machine; Forbes, B., et al; Ninth Annual Workshop on Microprogramming (MICRO-9) Proceedings; 1976; P 42-46

A specialized virtual system and virtual instruction set are described. The task defined is a real time software system used to control check sorting equipment. The mapping of the virtual system to the physical hardware is described and the design process illustrated.

MPL1700 A High(er)-level Microprogramming Language; Fisher, R. N., et al; Software- Practice and Experience; V7(1977); P 747-757

The report described the implementation of an intermediate level microprogramming language that is roughly equivalent to PL360. The design process and goals for a LISP-type interpreter are also described.

B. References Related to Hardware/Software/Firmware Analysis

Evaluation of hardware-firmware-software trade-offs with mathematical modeling; Barsamian, H.; Proceedings of the SJCC 1971; 1971; P 151-161

A mathematical model providing a rational basis for trade-off analysis is provided. Most of the factors affecting design choices are still current with some updating for cost factors that have changed since publication.

The interpreter - A microprogrammable building block system; Reigel, E. W., et al; Proceedings of the SJCC 1972; 1972; P 705-723

This is a discussion of work that lead to the development of the

Burroughs 'D-machine' interpretive system. Analysis of hardware and microprogrammed architectures is provided to determine a possible set of design choices in detail.

C. Instruction Set Requirements

Implications of Structured Programming for Machine Architecture; Tanenbaum, A. S.; CACM; V21N3; 1978; P 237-246

The study analyzed the use of a tuned instruction set for a C-like language. Stack oriented requirements and a sample virtual instruction set are described.

iMAX: A Multiprocessor Operating System for an Object-based Computer; Kahn, K. C., et al; Proceedings of the Eighth Symposium on Operating System Principles; Operating Systems Review; V15N5; 1981; P 127 - 136

The iAPX-432 (INTEL) computer uses ADA for the systems programming language. This article defines the hardware/software interface and requirements for ADA.

Supporting ADA Memory Management in the iAPX-432; Pollack, F. J., et al; Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems; SIGPLAN Notices V17N4; 1981; P 127 - 136

Instruction set requirements supporting ADA on the iAPX-432 are defined in this article. Specific discussion of ADA memory management for packages and block structures is also discussed.

INTEL 432 System Summary: Manager's Perspective; Manual 171867-001; 1981

This manual is a general discussion of the ADA-oriented Intel 432.

The GIBSON Mix; Gibson, J. C.; IBM Technical Report TR 00.2043, June 18, 1970 Computer Instruction Repertoire - Time for a Change; Church, C. C.; Proceedings of the SJCC 1970; 1970; P 343-349

The HP 1000 instruction set was modeled after the GIBSON mix.

D. Nanodata QM/1 References

The Nanodata QM/1 is a hardware system designed to support many different virtual systems and virtual instruction sets. Methods for handling virtual peripherals and I/O are also provided.

An Insight into PDP-11 Emulation; 9th Annual Workshop on Microprogramming (MICRO 9) Proceedings; 1976; P 20 - 26

An emulation of the PDP 11 as a virtual instruction set on the QM/1 is described.

A Meta-Assembler for Highly-parallel Microprogrammable Systems; Berglass, G. R.; 13th Annual Microprogramming Workshop (MICRO 13) Proceedings; 1980; P 181 - 189

This paper describes the emulation of the IBM System/370 as a virtual system/virtual instruction set on the QM/1.

Considerations for Local Compaction of Nanocode for the Nanodata QM-1; Rideout, D. J.; 14th Annual Microprogramming Workshop (MICRO 14) Proceedings; 1981; P 205 - 214

Compaction of microcode for a virtual instruction set implementation is described. An example is given from a C-type programming language.

E. Virtual Machine References

Architecture of Virtual Machines; Goldberg, R. P.; Proceedings of the NCC 1973; 1973; P 309 - 318

The concept of a 'hardware virtualizer' that permits concurrent execution of multiple virtual systems is described along with the necessary data structures and hardware/firmware support.

Evolution of a Virtual Machine Subsystem; Hendricks, E. C., et al; IBM Systems Journal; V18N1; 1979; P 111 - 142

VM/370's development, from the early CP-67 until 1979, is traced in this article. Providing data communications, virtual machines, and support for existing operating environments were problems encountered and solved.

VM/370 - A Study of Multiplicity and Usefulness; Seawright, L. H., et al; IBM Systems Journal; V18N1; 1979; P 4 - 17

A very good overview of the usefulness for virtual systems software is provided in this reference work.

F. Firmware Supported Languages

These papers discuss the implementation of various languages as firmware virtual instruction sets.

A Firmware APL Time-sharing System; Zaks, R., et al; Proceedings of the SJCC 1971; 1971; P 179 - 190

An APL Emulator on System/370; Hassitt, A., et al; IBM Systems Journal; V15N4; 1976; P 358 - 378

APL\3000 Issue of HP Journal; V28N11; 1977

APL Now Available on a Small General Purpose Computer; HP Computer Advances

Hardware Assisted Virtual Instruction Set Implementation

Microprocessor Implementation of Mainframe Processors by Means of Architecture Partitioning; Agnew, P. W, et al; IBM J. R&D; V26N4; 1982; P 401 - 412

> This article describes the implementation of the IBM 370 unpriviledged instruction set using M68000 chipsets. This is a technical description of the investigations that led to the IBM XT/370 and AT/370 products.

An Analysis of C Machine Support for other Block-Structured Languages; Hill, D. D.; ACM Computer Architecture News; V11N4; 1983; P 6 - 16

Additional virtual instruction set implementation references can be found in the discussions of the RISC/CISC (Reduced Instruction Set/ Complex Instruction Set) literature such as:

The 801 Minicomputer; Radin, G.; Proceedings of the Symposium for Programming Languages and Operating Systems; ACM SIGARCH Computer Architecture News; V10N2; 1982;

also published in: IBM J. R&D; V27N3; 1983; P 237 - 246

> The 801 project is the IBM development from which major members of the HP SPECTRUM Project came from.

Architecture of SOAR: Smalltalk on a RISC; Patterson, et al; 11th Annual International Symposium on Computer Architecture; ACM SIGARCH V12N3; 1984; P 188 - 197

> A very good paper describing the implementation of a particular instruction on a RISC architecture.

The reader is referred to the ACM SIGARCH publication Computer Architecture News for an analysis of the RISC/CISC tradeoffs. This is currently an area of considerable debate within the academic and technical community.

II. ADA References

The reader is directed to the AdaTec letters in general, published by ACM, and in particular the Proceedings of the AdaTEC Conferences (annually).

Is ADA Too Big? A Designer Answers the Critics; Wichmann, B. A.; CACM V27N2; 1984; P 98 - 103

> An examination of the features of ADA that are useful - and those that are discardable.

ADA: Past, Present, Future; Ichbiah, J.; CACM V27N10; 1984; P 990 - 997

This interview with Jean Ichbiah presents a good 'feel' for the thought process of those who architected ADA.

STONEMAN - Requirements for ADA Programming Support Environments; US DoD; 1980

This document defines the KAPSE/MAPSE/APSE environments. Later documents have expanded, clarified, and refined the concepts, but this document is still 'all in one place'.

ADA Programming in the 80's; IEEE COMPUTER; V14N6; 1981

This issue of COMPUTER discusses a range of ADA concepts and issues: history, environments, and validation; and provides a background for readers new to ADA.

Reference Manual for the ADA Programming Language; DoD; 1983

This is THE standard - MIL-STD-1815a. The ANSI/ISO standards both reference this work as definitive.

ADA Software Tools Interfaces; Proceedings of the Bath Workshop; 1983

The proceedings examines the use of the DIANA intermediate language as a representation for ADA programs.

There are a large number of articles, spread widely in the literature, discussing ADA as PDL - Program Design Language. These articles describe the use of ADA as a high level 'pseudo-code' in which program and system specifications can be described.

ADA Joins the Army; Defense Electronics V15N12; 1983; P 68 - 79
First ADA Compilers Show Diversity; Defense Electronics V16N3; 1984; P 52 - 63

These articles discuss the variety of implementations of ADA compilers with an overview of some of the structures and approaches.

III. HP Product Related References

Most of these references come from the HP Journal, a publication of HP, which can be ordered from:

Hewlett-Packard Journal
3000 Hanover Street
Palo Alto, CA 94304

Each issue deals primarily with a single topic and will be cited as such with some annotation. The citations to a month and year refer to a particular issue of the HP Journal.

A. HP 1000 References

OCTOBER 1971
The HP 2100A microprogramming is described in this issue.

JULY 1972
Microprogramming using WCS is discussed.

MARCH 1977
The micrprogramming for the E-Series is discussed.

OCTOBER 1978
This issue includes a description of the microcode F-Series instruction sets.

OCTOBER 1974
The microprogrammability of the M-Series is discussed.

FEBRUARY 1975
Microprograms for the FFT are discussed.

(undated) 1984
Issue on A900 implementation and firmware.

B. DESKTOPS

HP 9836

MAY 1982
Discusses the architecture and software of the HP 9836. Small section on MODCAL is presented.

HP 64000

OCTOBER 1980
Discussion of machine independent HP PASCAL compiler on the HP 64000.

C. HP 9000

AUGUST 1983 and MARCH 1984
These two issues provide considerable detail in the implementation of the HP 9000 hardware and software architecture, including the chipset capabilities. The March 1984 issue discusses the SUN OS kernal.

A 32b VLSI CHIP; Beyers, J., et al; Digest of Technical Papers of the 1981 IEEE ISSC Conference; (1981)
An NMOS VLSI Process for Fabrication of a 32b CPU Chip; Mikkelson, J., et al; Digest of Technical Papers of the 1981 IEEE ISSC Conference; 1981
These articles are the first public descriptions of the FOCUS chipset.

D. HP 300/3000

HP 300

JUNE 1979

The HP 300 is the same hardware as the HP 3000/30/33 systems. Different CPU microcode is in use. At one time the HP 3000/4X CPU was also considered for microcoding of the HP 300 instruction set.

HP 300 Computer System Architecture

This HP manual provides one of the best descriptions for any HP computer system architecture.

HP 3000

HP 3000 Computer System

JANUARY 1973
This issue discusses the HP 3000 Computer System - the original.

HP 3000 Series II

AUGUST 1976

This issue discusses the Series II in general terms.

Microprogramming in an Integrated Hardware/Software System; Sell, J.V.; Computer Design; 1975; P 77-83

John Sell has since gone on to Ridge Computer Systems and implemented the Ridge CPU using a semi-RISC approach.

The microcode was shipped as listing along with the manual set for most HP 3000 Series II systems.

HP 3000/30/33

SEPTEMBER 1979
The HP 3000 was implemented in a totally different hardware packaging from the, then larger, HP 3000 Series III. The instruction set stayed the same, but the hardware completely changed.

HP 3000/68

MARCH 1982
The microprogram development for the HP 3000/68 is discussed in this issue.

Also of interest to HP 3000 readers may be:

Architecture of the HP 3000; Kell, J.; HP IUG Journal V5N1-2; 1982

This discussion includes the layout of the Series II microcode words.

The MPE IV Kernal: History, Structure, and Strategies; Busch, J. R.; Proceedings of the HP 3000 IUG Conference in Orlando; 1981

Some external documentation of the MPE-V/E firmware changes are also in publication.

1003. THE DESIGN OF APPLICATION UTILITIES FOR IMAGE

Don LaRobardiere
C & L Systems, Inc.
1250 E. Ridgewood Ave.
Ridgewood, New Jersey 07450

1. Introduction

Today's expense of developing custom software for dedicated applications warrants a further look at alternatives. This paper will attempt to outline the major stumbling blocks in developing application software, and what alternatives there are to avoiding them. Several references to an existing set of utilities developed since the late seventies to solve these types of problems will be referred to in this paper.

1.1. What Kinds of Applications Involve Image

Since its inception in the mid 70's HP's IMAGE has been the focal point for most users of data bases. IMAGE has grown in size and complexity in an attempt to meet more of the users needs. User applications have grown as well. Typically their solutions involve one or more of the following approaches to storing and retrieving data. Lets consider them.

1.1.1. The On-line Multi-user

This approach to supporting online multi-users is quite typical of the manufacturing environment. Here lots of terminals are tied in with a multitude of users gaining access at different levels. Typical requirements usually do not require a lot of data but many records are frequently accessed. Response varies and usually is in tens of seconds.

There are quite a few packages with a commercial flavor that address this area. They are commonly referred to as application code generators, data entry and retrieval subsystems, etc.

1.1.2. Real-time processes

Another approach to the data-base usually involves a high speed programmatic interface. More often than not this is custom code that has been optimized for I/O performance. Such types of coding are usually done for data acquisition packages. Typically IMAGE is used here for long term storage of condensed data, rather than an online data base for obvious performance reasons.

1.1.3. Batch Operations

One of the oldest approaches is the batch approach. Typical of this are reports that require a sufficient amount of time to complete. Usually these are run in off-hours so as not to disturb any real-time or multi-user activity.

2. What are the Common Pitfalls in These Approaches

We have just briefly mentioned three generic approaches most applications fit in. However, most applications are rushed into without a lot of forethought in the planning stages. It is difficult to design from scratch and come up with an acceptable solution. Let us remember that "difficulty is an excuse history never accepts" and consider the following "gotchas" the next time we have a custom application to solve.

2.1. Inadequate Specs

One of the prime reasons for failure is the lack of an adequate functional specification. Without it, although the design may be good, the objective for doing the project may be missed. The functional spec should be written in terms the end user can understand. Only when both the customer and designer mutually concur on the functional spec does the actual design phase begin.

Typically the design spec should be done by software designers and not just programmers. The design spec is not meant to be readable by the inexperienced software type but must be written so it is thorough and concise. Typically, all tables, file layouts, forms, software module interfaces, module descriptions, etc. are included in this spec.

After the design spec, the designer(s) should write a preliminary draft of the user and program reference manuals for the customer to check over. This will further inforce-enforce the objectives and catch minor deficiencies in the functional spec.

Only after the functional and design specs, and the preliminary manuals are done, are we ready to involve the programmer.

2.2. Poor Design & Coding Methods

Everything falls from the structure laid out in the design spec. Just like nobody in their right mind starts out to build an expensive house without adequate plans, neither should you begin coding without a good design spec in hand.

Programmers should be forced to follow structured methods that reinforce the ideas of modularity. choosing the proper language helps but does not ensure that you will end up with a structured and supportable system.

Often the best language is not the one touted at the moment by all the media but one that has been around a long time and is well proven. I prefer Fortran 77, like HP's RTE it has evolved and been tested in thousands of real-life applications. Its code is the most efficient of any structured language on the 1000, and it's the most portable in terms of input/output interfacing, something the current university taught languages seem to ignore.

The proper approach to writing a module is to create a flowchart outline that speaks in terms of what is to be done not how to do it. It should probably even make sense to the customer. After all the modules have been flowcharted in this manner, then the coding can commence.

From the structured approach, key aspects like Modularity, Data Flow, Code Efficiency, Flexibility, and Portability evolve naturally.

2.3. Inadequate Testing

Lets assume we have a good set of specs and competent programmers who have created this masterpiece of software. They have claimed it's working. From their viewpoint it probably does what they have asked it to, but the real test comes when different users start "banging" on the code.

Beta Test sites have been used for years to shake down new software. There is no hard and fast rule as to what is and isn't enough, but experience shows that when the bugs stop coming in, either nobody's using your software or you've invented the perfect program (hardly likely).

Most developers have a level at which they can support a certain number of bugs with an adequate response time. It is at this time that the software is usually turned over to the customer.

2.4. Inadequate Customer Instruction

Probably the biggest reason why custom software or software packages don't succeed is the misunderstanding by the customer of what the package can and cannot do, as well as how to get it to do what he wants it to.

Easy to understand instruction manuals for the beginner and the advanced user are a necessity. They should have numerous illustrations, an index, a table of contents, and not be too wordy. Many examples seem to work the best in helping the customer learn to do what he wants.

Many companies today also provide on-line help facilities that can be quite extensive; even bordering on the fringes of Artificial Intelligence. Several have directories of keywords that are laid out in a hierarchical fashion to provide several levels of increasing explanations. Some even display a page of help for the task the user is currently working on without the user having to go searching for it.

In addition, a very friendly easy to use command interface helps the customer feel confident quickly. Characteristics of such "friendly" interfaces are easy to remember commands, a command structure that can be miss-typed yet understood by the command interpreter, prompting for missing parameters, informing the user of incorrect parameters, and re-execution with/or without modification of what was entered; these are just some of the ways to make the interface friendly in the traditional sense.

2.5. Inadequate Software Support

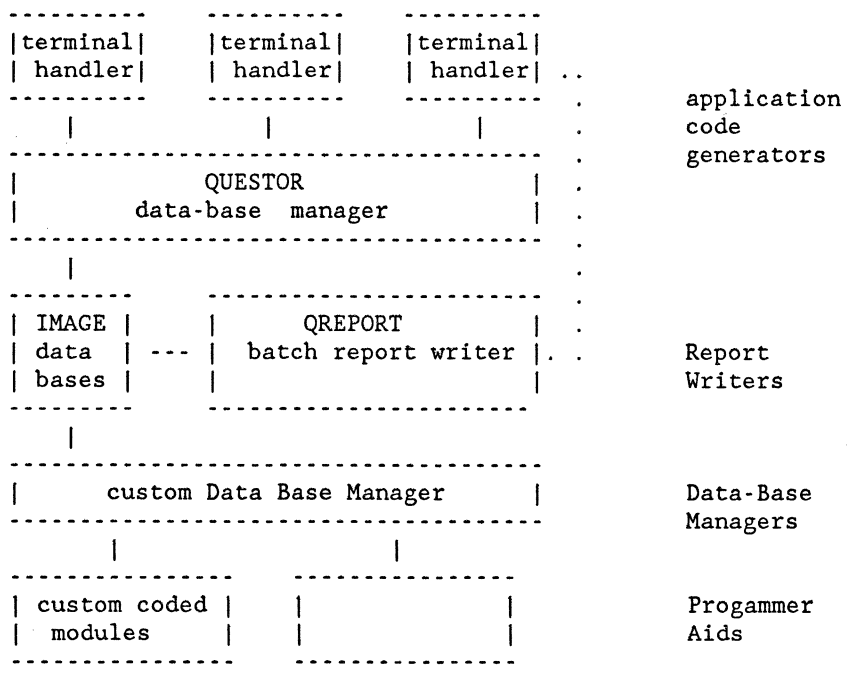
Many software projects come and go, those who stay have support. We have observed that support depends on primarily one thing, people. Especially those who did the design, coding, and debugging. If they aren't around anymore watch out. Even the best documented code can be a support problem for new people.

Usually the smaller the development team, the quicker the response to your bugs.

This is generally because those few people know more of how everything works, fits together, and have the skills at hand to implement the changes almost immediately. In larger corporations where the development teams are larger, and the turnover rate greater, the support people are less familiar with less software and usually less enthusiastic about your problem. The reason for a lot of this is large companies tend to have a separate staff of people for support. If it was your software causing the problem I'm sure you'd feel more responsive to fixing the bug. 3. What Can be Done to Ensure a Success

It has often been said that "Experience is a good school, but the fees are high". We all are constantly being re-educated in our way of doing things. Lets consider the following ways to attack your problems. I'm sure at the end you'll agree that the "fees" are much less than those of the school of "Hard Knocks" ...

The three data-base access approaches mentioned before are illustrated in the following pictorial.



The approaches are outlined in sectional blocks to depict various module groupings; Application Code Generators, Report Writers, Data Base Managers, Custom Code. Several of these groups represent commercial offerings on the market place. Considering the features of each should make us more hesitant in re-inventing the wheel next time we have a project to consider.

3.1. Application Code Generators

Application Generators have evolved over the last ten years primarily. They have become easier to use and more comprehensive in what they can and can not do. Here is a list of features supported by one of our products, QUESTOR, which I think is state of the art since it eliminates the intermediate phase of having to compile the generated code and link it.

Should be quick and easy to implement. One should not have to get into checking code that has been generated or linking it. We have found that the customer's confidence in a product is greater when in a matter of minutes you can generate an operational form on his data base.

The package should support individual logon accounts, menus, function keys, help screens. All of these should be customizable without code by the user directly.

Support of multiple data bases in various states of activity while concurrent development with the package has also been found to be most useful in almost all production environments. In addition the ability to make use of the new backup and recovery procedures of IMAGE II is quite essential.

The question of concurrent access to the same record is easily answered since record locking is provided as a function of the package. The second user is still allowed to examine but not to change the record. Access levels of update, add, and delete capabilities are assignable on an individual account basis.

The package should not be a system resource "hog". It needs to co-exist with many other packages already out there. It's requirements on SAM, memory, driver support, and operating system types should all be minimal and transparent. Any package that takes advantage of the niceties of RTE will find themselves in trouble in the future and so will you if you're using them.

QUESTOR, besides being an application generator, can provide a user interface shell for other programs. A convenient "command transfer file" approach has been implemented to pass data to and from other programs running within this shell.

3.2. Report Writers

One of the programs that can run under the QUESTOR shell is QREPORT, a procedure file driven report writer. Like many other report writers common features are; the ability to generate elaborate reports involving horizontal and vertical tabulation, totals and subtotals, multiple different format types for data, sorting at various levels by items, etc.

What is particularly nice about them is that you don't have to write any code. By various procedure files passed you can generate a multitude of reports. Under the QUESTOR shell you can even pass information from the form to further qualify the search for data.

In addition several hooks have been allowed where custom code can be added for special formatting purposes.

3.3. Data Base Managers

The QUESTOR package has its own "data base manager" for handling the various terminal handlers as depicted in the previous figure. In general data base manager packages provide a more general function. Support of custom code is generally their role. Some of the benefits of using them follows.

All the data base access code is centralized in one module. This does not cause a proliferation of the data base access routines in each module dealing with data from the data base. Supportability and conservation of memory are the two key benefits here. Another usually is support of larger data-bases.

Another feature that is usually implemented is a record locking mechanism; something that IMAGE doesn't provide directly.

Startup and shutdown of the data-base is more controlled and easier to support. Automatic bootup features, like QUESTOR's, can make customer support a lot easier.

The development of custom code generally progresses faster since programmers do not have to get involved in the details of the data base operations.

3.4. Form Generators & Programmers Aids

Concerning custom code, a number of programming aids come to mind. With packages you can concentrate more on your application than on how to make the system work for you.

If you are using forms on the HP1000, many of you may have experienced the nightmare of trying to get them to work on one terminal, one driver, and under one operating system. Try to imagine supporting forms on the gamut of HP's operating systems, drivers, and crts.

Over the past several years we have survived with our forms generation package, QFORM. QFORM has more than most of the existing form packages today since it was developed around 1980 for the industrial process control market, where operator interfacing is very crucial. It can considerably cut down code developement to a few lines. Lets look at the benefits of such packages.

Format conversions, programmatically controlled, allow multiple use of the same form field. The HP formatter is not used so an additional benefit is a saving of approximately 5k of code.

Ideally the packages should provide all the terminal support required to get your application done. Function key support and help screens are a necessity.

Equally important is the minimal use of system resources. The package should only have to use a few thousand words of code on top of your program. It should not force you into using CDS, EMA, extensive amounts of SAM, or additional programs.

If you intended to port your code to other HP systems, or even intend to run your application on a different terminal or driver, portability becomes a key question since the packages are so closely tied to the I/O of the operating

system. Support becomes a key question to ask in this area.

4. Conclusion

We have pointed out several areas in which code developement and success fail. We have tried to illustrate, based on our own experiences of developing products, what we felt was required to accomplish a variety of applications based around IMAGE.

Let us conclude then with what we feel is "The Good-Programmers Seal of Approval Checklist" for successful software projects.

Good Functional & Design Specs should provide for:

- Adequate flexibility to cover unexpected customer desires
- User friendliness
- Portability
- Ease of implementation

A Good Code Implementation is based on:

- Use of a structured language with a structured approach !
- Proper coding techniques
- Proper documentation before and after the project

Survivability of the Code is based on:

- Ease of installations
- Current support levels

1004. CAR/1000
Computer Aided Calibration and Recall System

Leonard Singer
Thomas L. Barrett
Information Systems Software
General Electric Company
Re-Entry Systems Operation
3198 Chestnut Street
Philadelphia, Pennsylvania 19101

1. Introduction

Precision, accuracy and quality are the key words in the calibration laboratory at General Electric's Re-Entry Systems Operation. Here, technicians calibrate and maintain sophisticated measuring instruments whose accuracy is vital to many departments. Thermal converters, voltmeters, generators, pressure gauges and temperature chambers are just some of the thousands of pieces of equipment that must be calibrated regularly.

In the late 1970's, however, the calibration laboratory had problems. The staff was overwhelmed by the paperwork needed to track the status and location of every instrument inside and outside the laboratory. The complexity and size of the process outstripped the ability of the technicians and support staff to respond to and maintain this manual record keeping system. A solution had to be found. The solution chosen was to develop our own on-line (real-time) data base management system that would meet user needs through 'fixed' application programs and related database techniques.

In 1980, when this system was first envisioned, we knew we wanted a system that would provide:

- o Real-time, interactive data collection and feedback compatible with existing terminals, printers and our HP1000 minicomputer system.
- o Analysis routines to automate automatic scheduling and other work control functions.
- o Communication packages to interconnect with other property systems maintained on a Honeywell mainframe.

At that time, we knew of no commercially available package with such features. Therefore, we did what we felt we had to do. We designed, developed and installed our own system that was 100% compatible with our existing equipment resources and mimicked the 'proven' manual system. But because of intense pressures for a viable (and quick) solution, almost all of the normal software development cycle was short circuited and NO untried approach to this manual/automatic conversion was attempted.

The resulting CAR/1000 system was partially on line within 60 days and fully operational 120 days after the start of initial design tasks. We had our problems,

but through some innovative use of IMAGE and GRAPHICS we survived. In fact, the 'CAR' system of today is virtually the system that was specified back in 1980 and it continues to meet the everyday and long term needs of the calibration laboratory.

2. Calibration Laboratory Facility

The current calibration laboratory facility is divided into two sections, one for mechanical equipment and one for electrical equipment. Each section has three specific work areas:

(1) Window area for receipt and disbursement of equipment.
(2) Holding area for storage of equipment scheduled for or recently completed with calibration.
(3) Maintenance and calibration area where the work is carried on.

Equipment constantly moves between these laboratory areas and other departments. Knowing the location and/or status of any individual item of equipment is vital to providing quality service. With the CAR system, the company can keep accurate account of thousands of pieces of mechanical and electrical equipment as they cycle into and out of the laboratory for maintenance.

3. The CAR/1000 Package (Features)

CAR is an interactive, real-time system that a user accesses to get up-to-date information about a specific piece or family of equipment, by just selecting from a menu of options.

The system helps different people carry out their responsibilities more effectively. The calibration laboratory supervisor relies on CAR:

(1) to keep track of the status and location of each instrument as it progresses through the laboratory.
(2) to schedule calibration work activities and assign tasks to technicians.
(3) to identity questionable equipment for retirement and/or major overhaul.
(4) to provide equipment receipt and pickup information-gathering functions.

The calibration laboratory technician relies on CAR:

(1) for the identification of assigned work.
(2) for listing of test equipment necessary for calibration tasks.
(3) to enter time and material associated with each calibration function.

The equipment owners rely on CAR:

(1) for ALERT notices of due, past due and early warning equipment listings.
(2) to determine equipment calibration status and when items are ready

4. The CAR/1000 (Specification)

A diagram of our CAR/1000 system is shown in Figure 1. Some points of interest are:

- o FORTRAN applications were segmented and only very loosely tied together via common. A menu segment retrieves the proper work segment per user menu response.
  This segmentation allowed us to write a very large package and have it reside within our maximum 28 page partition.

- o Problems associated with who can or cannot update the information about a specific piece of equipment were handled via a table of operations and pointers back through the segment via an HP routine.

- o GRAPHICS/1000 was the selected method of providing graphics based on reliability and tolerance information. FORTRAN applications using the GRAPHICS package were again segmented.

- o This system was totally compatible with a communications network (Gandalf) recently installed at our facility. Using this network and its associated DEC terminals we were able to communicate from widely dispersed locations to our HP1000 mini system.

- o System analysis and maintenance are accomplished using COMSCI's IMF/1000 package. This package provides for quick on-line capacity changes and relinking of specific records or entire data sets.

Our computer system then (in 1980) and now is an M series HP1000 minicomputer using RTE-IVB operating system. The hardware diagram is shown in figure 2.

5. The Data Base Structure

As shown in Figure 3, the CAR data base design is extremely simple. Keyed items (those that provide for quick access to data residing in any one of the six detailed data sets) were limited to these:

- o Inventory Control Number...A number assigned to each item of equipment upon receipt at Re-Entry Systems. This number is also the key to the existing property system.

- o Family Code.....A number which groups like pieces of equipment. Such grouping is extremely important when providing statistical operational data. The family code is a 6 digit numeric code assigned to each item upon receipt.

- o Calibration Due Date.....The date when a specific item of equipment is due for calibration and/or maintenance. Keying on this number provides for quick extraction of items due on any specified week.

- o Calibration Status..A numeric code representing the calibration

status of a specific item of equipment. Keying on this value provides for quick extraction of any of the 8 status conditions.

(0) Not in Calibration Laboratory
(1) Scheduled for calibration
(2) Scheduled but waiting on parts
(3) In storage
(4) Calibration complete
(5) At vendor
(6) Out-er...The code of the operation using the item. Keying on this value provides quick listings and summaries for all levels of management.

- Rack Identifier....The rack where a quantity of inventory controlled equipment resides. Keying on this value provides an easy way of entering all equipment associated with a particular rack.

As with everything in life, nothing is really free. The payment for quick access to CAR via these seven elements requires that each modification physically rewrites the entire data entry.

6. Applications

Access to this CAR/1000 system is through a twenty-nine segment FORTRAN application program. This program (See flow diagram, Figure 4.) uses the calls provided by FORTRAN/IMAGE to satisfy customer information entry and gathering requests. The interesting features of this application include:

- Session hello files to provide access into the system. These hello files pass information into the program via an HP routine.

  Data passed include:

  (1) Logical unit number of requesting device (terminal).
  (2) Logical unit number of printing device.
  (3) Code reflecting who the user is.
  (4) Code reflecting valid operation number.

- Internal menus to provide requested actions. Users need not know how the system is constructed since menus direct user to the proper segment. (See Figure 5, 6, 7, 8 and 9 for menu displays.)

Reliability and out-of-tolerance data accumulated within the CAR/1000 data base are further compiled for reports and display by a nine segment plotting program. Data generated by this application are used by both our calibration laboratory and our customers to determine the effectiveness of the calibration tasks. Sample plots of reliability, out-of-tolerance and operation levels are shown in figures 10, 11 and 12.

To further identify those items causing a low reliability or a high out-of-tolerance condition, another set of FORTRAN applications was developed. Using these

applications, calibration laboratory personnel identify and may eliminate or overhaul problem-causing items.

Yet another FORTRAN application was required to develop management ALERT reports from the data base; these reports identify items past due or due for calibration. The summary data, along with equipment listings, are sent (monthly) to operation managers and our customer iences. One of the biggest problems was the lack of good development and design documentation. Even today we still hurt a little when we attempt to modify the existing code. If we had to do it all over again (with a little more time), we would demand a complete set of documentation.

- o Requirement Specification
  - o Software Requirements Document
  - o Design Document
  - o Test Plans and Result Documentation
  - o User's Manuals
  - o Installation Manuals
  - o Full and complete Programmers Design Notebooks

Other problems uncovered include these:

- o The data base was opened at the front end of the application program, causing (at times) the unavailability of the data base to other users. We should have opened the data base just prior to a function then immediately closed it. This would have provided some contention for the limited data base open level.

- o The HP-provided lock routines tended to slow down the system. A user-friendly lock and unlock system had to be designed.

- o Updates and writes to the data base were designed into the appropriate segment. This function should have been handled via a 'father-son' program (schedule without wait and allowing for a queue). This arrangement would have speeded up transactions.

- o Data base design task should have allowed for additional time to completely analyze each of the required elements. Some are incorrectly typed, others are superfluous.

7. Summary

The benefits of the CAR/1000 system to our operation have been significant. For now, our management, supervision and technical staff have full access to all of the calibration laboratory inventory and work records. This has meant increased productivity within the laboratory, lower quantities of past due equipment on the shop floor, elimination of most paper records, improved data integrity, improved historical and service files, enhanced workload management, means for immediate response to inquiries from equipment users, improved equipment turn-around times. All of these add up to timely, accurate information by which to manage our calibration facility.

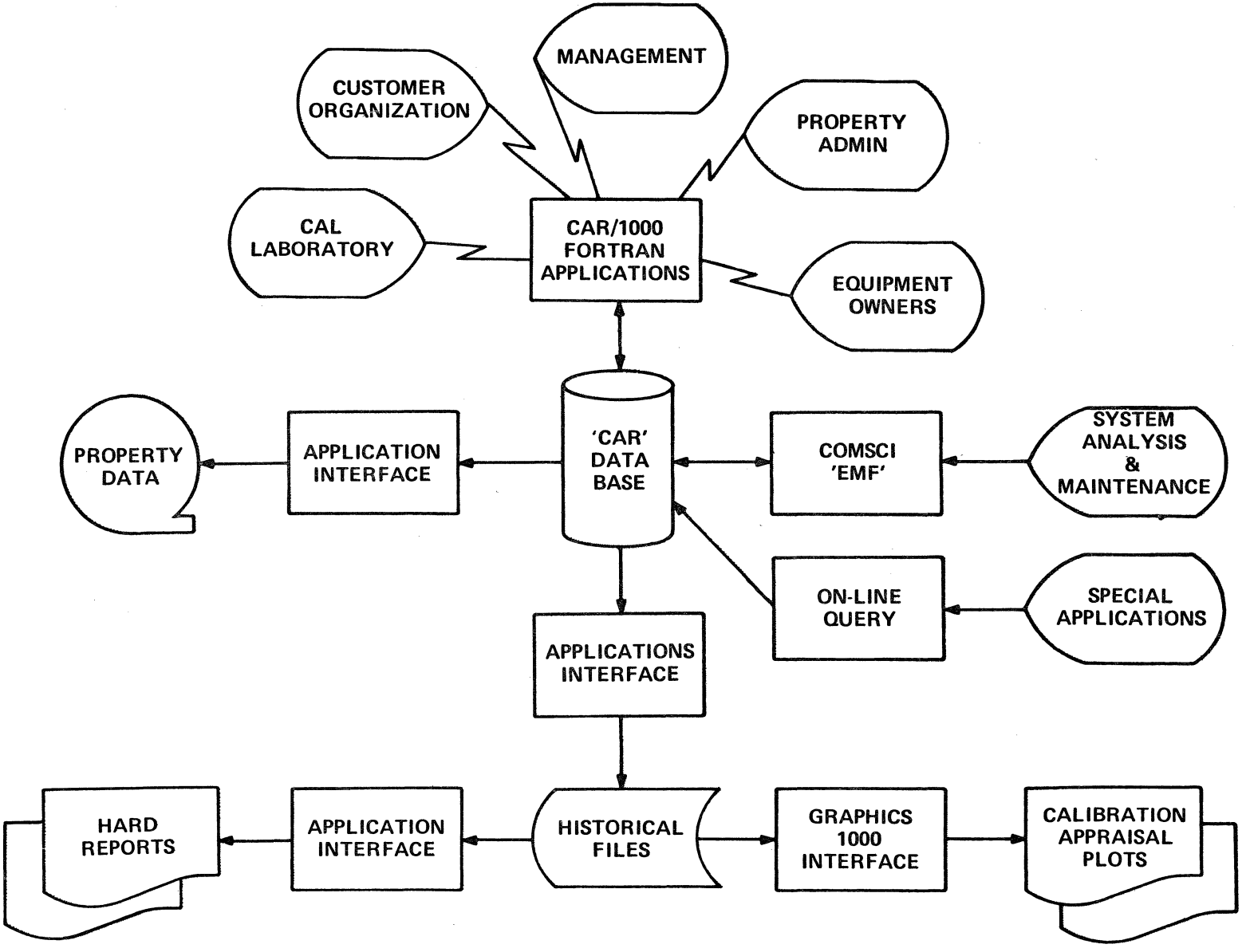


Figure 1. System Block Diagram CAR/1000

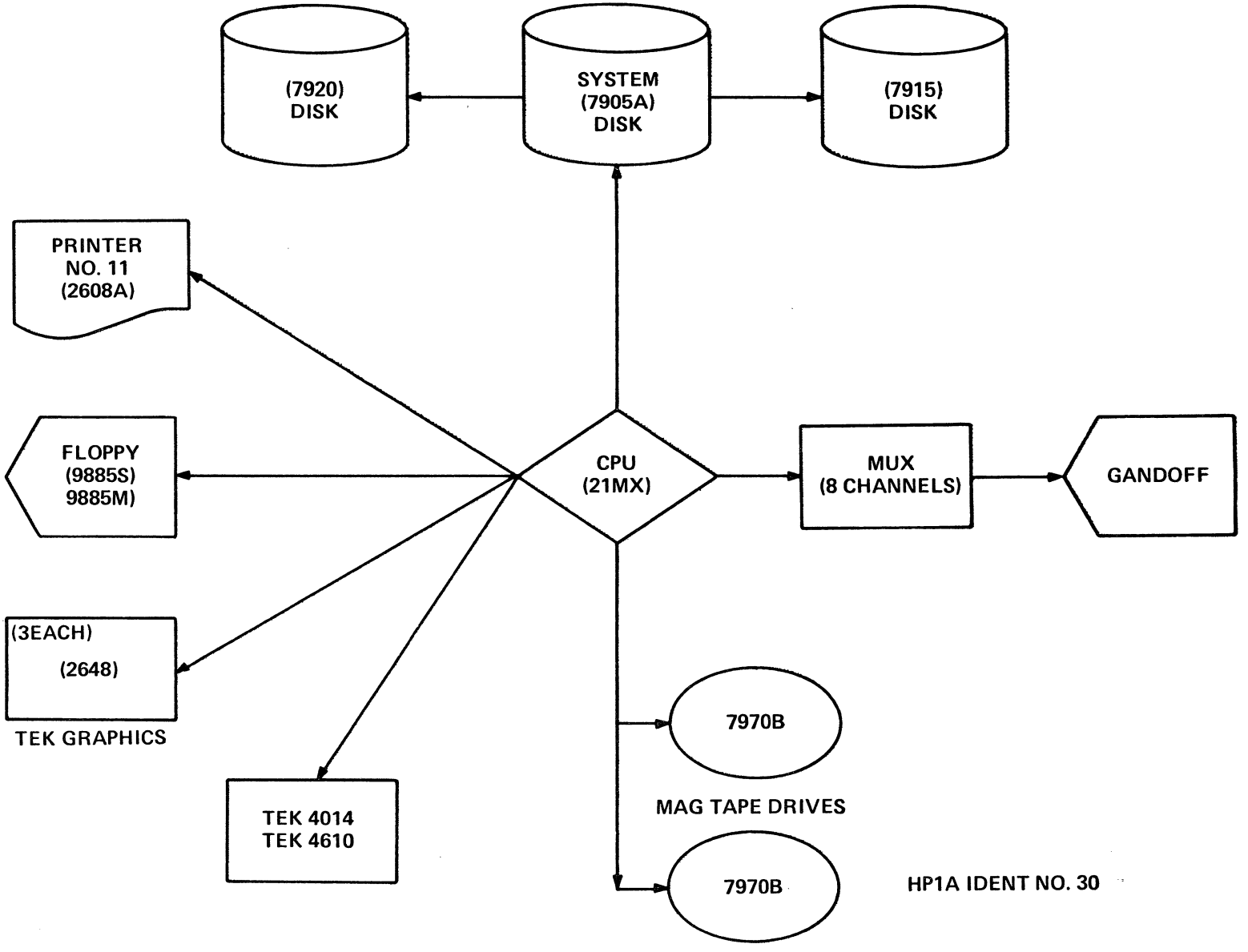


Figure 2. Hardware Configuration

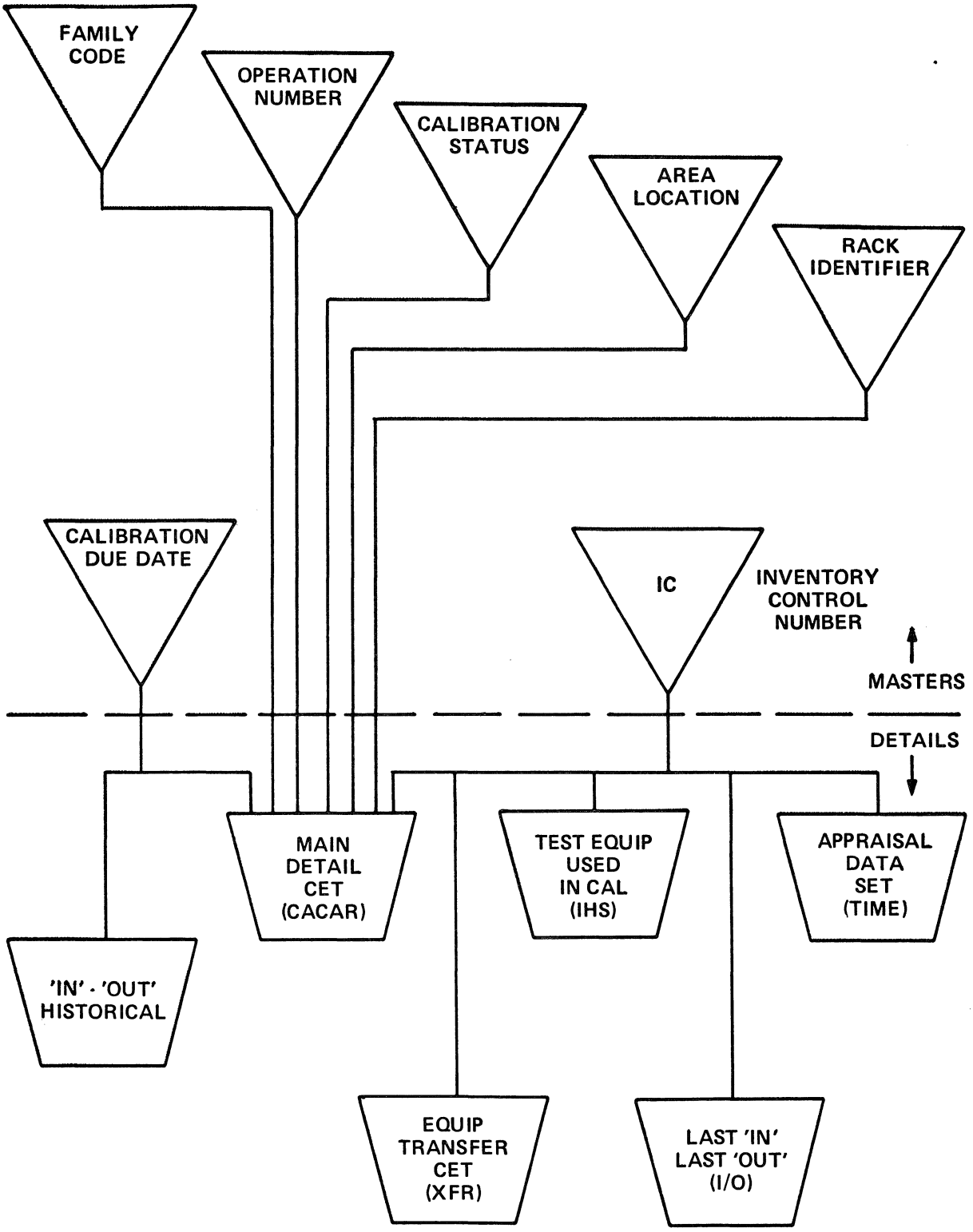


Figure 3. CAR/1000 database structure

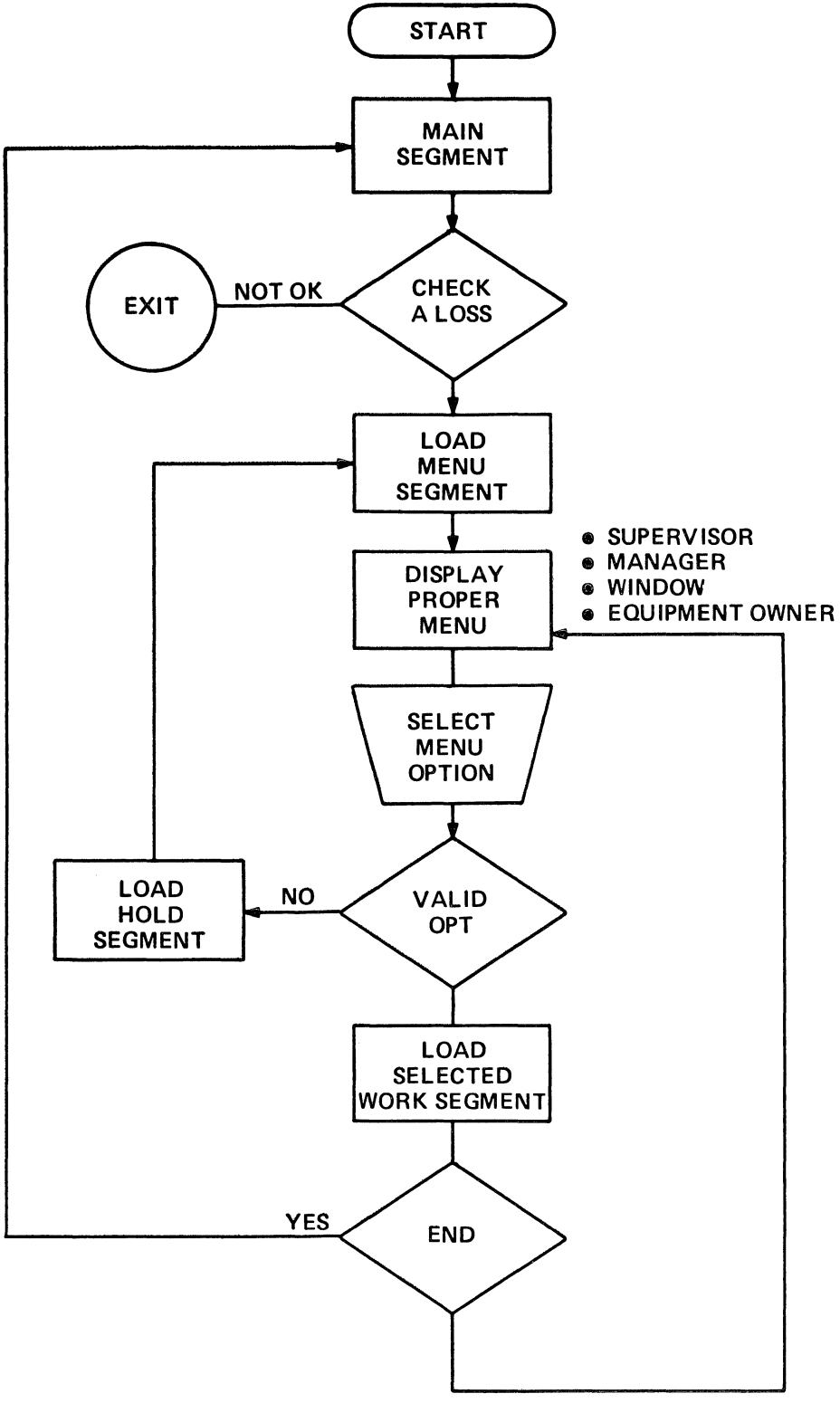


Figure 4. CAR Application Program Flow Diagram

```
Computer Aided Calibration and Recall System
============================================


MANAGEMENT 'CAR' MENU

ENTER 'HELP' TO ACTIVATE HELP FILE

SCREEN NO:                    GENERAL FUNCTION:
( 1) THIS WEEKS STATUS....IN/OUT (FOR PENDING WEEK) SUMMARY
( 2) WORK LOAD SUMMARY....CAL LAB PREDICTED LOAD SUMMARY
( 3) MAINTENANCE SUMMARY..PASS/FAIL/REPAIR HISTORY
( 4) EQUIPMENT SUMMARY....EQUIP STATUS BY OPERATION
( 5) PAST DUE SUMMARY.....PAST DUE QUANTITIES BY SECTION
( 6) TRANSACTION SUMMARY..NEW EQUIP & TERMINATION ACTIVITY
( 7) SERVICE CONTRACTS....LIST OF SERVICE CONTRACTS AND DUE DATES
( 8) PAST DUE EQUIP LIST..LISTING OF EQUIP BY OPERATION
( 9) SEARCH BY IC.........LISTING OF SELECTED IC ITEM
(99) END MENU/TERMINATE PROGRAM
SCREEN?99


     SAMPLE SCREEN DISPLAYS ARE PROVIDED ON FOLLOWING PAGES
```

Figure 5. Manager's CAR/1000 menu display

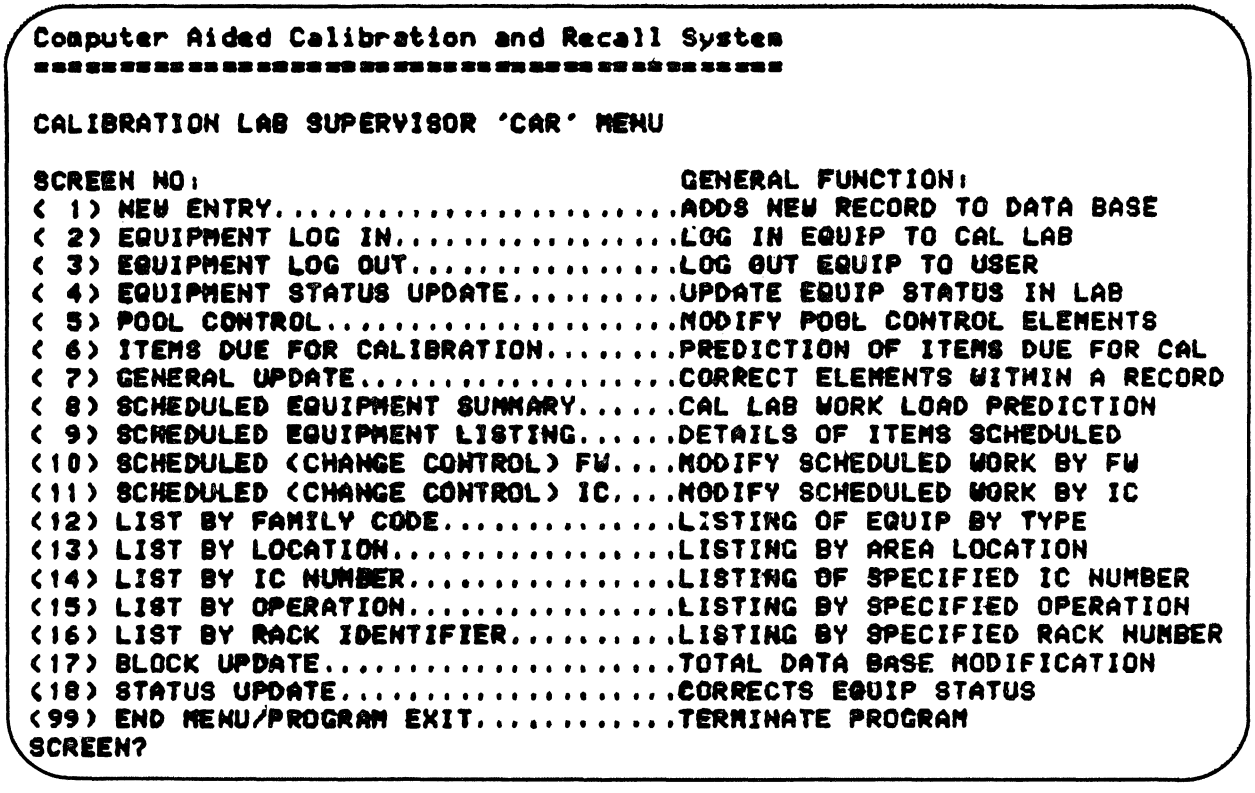


Figure 6. Supervisor's CAR/1000 menu display

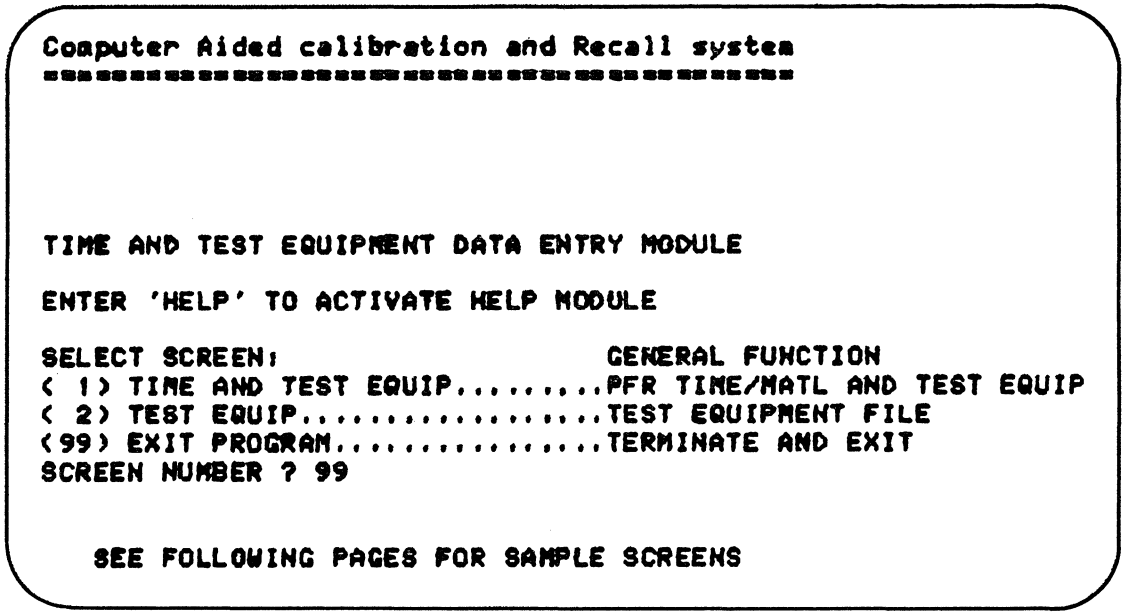


Figure 7. Technician's CAR/1000 menu display

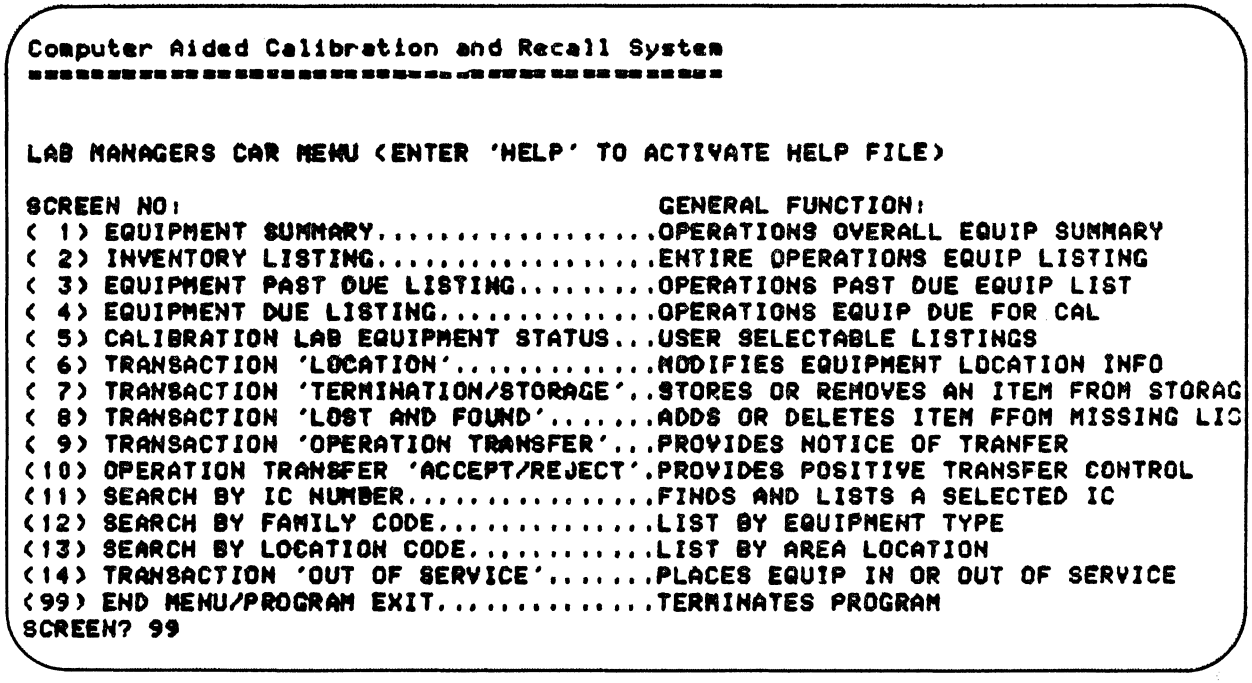


Figure 8. Equipment Owner's CAR/1000 Menu Display

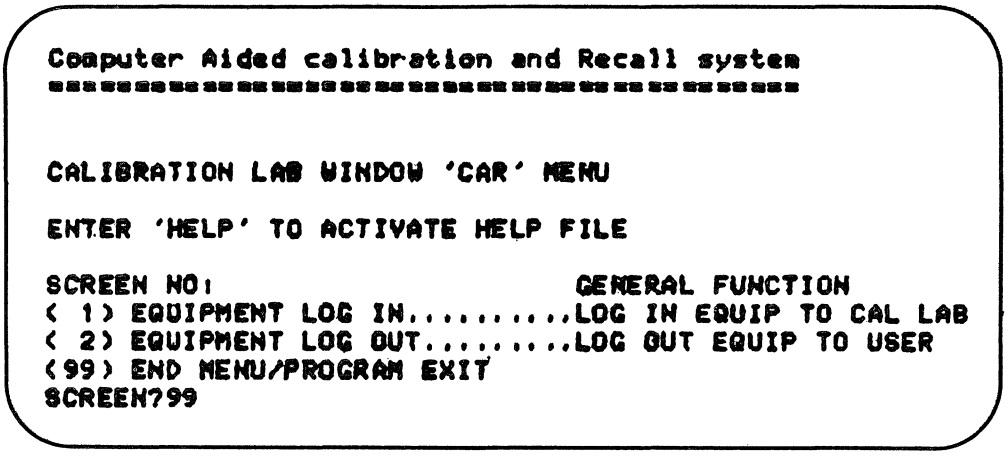


Figure 9. Laboratory Window CAR/1000 Menu Display

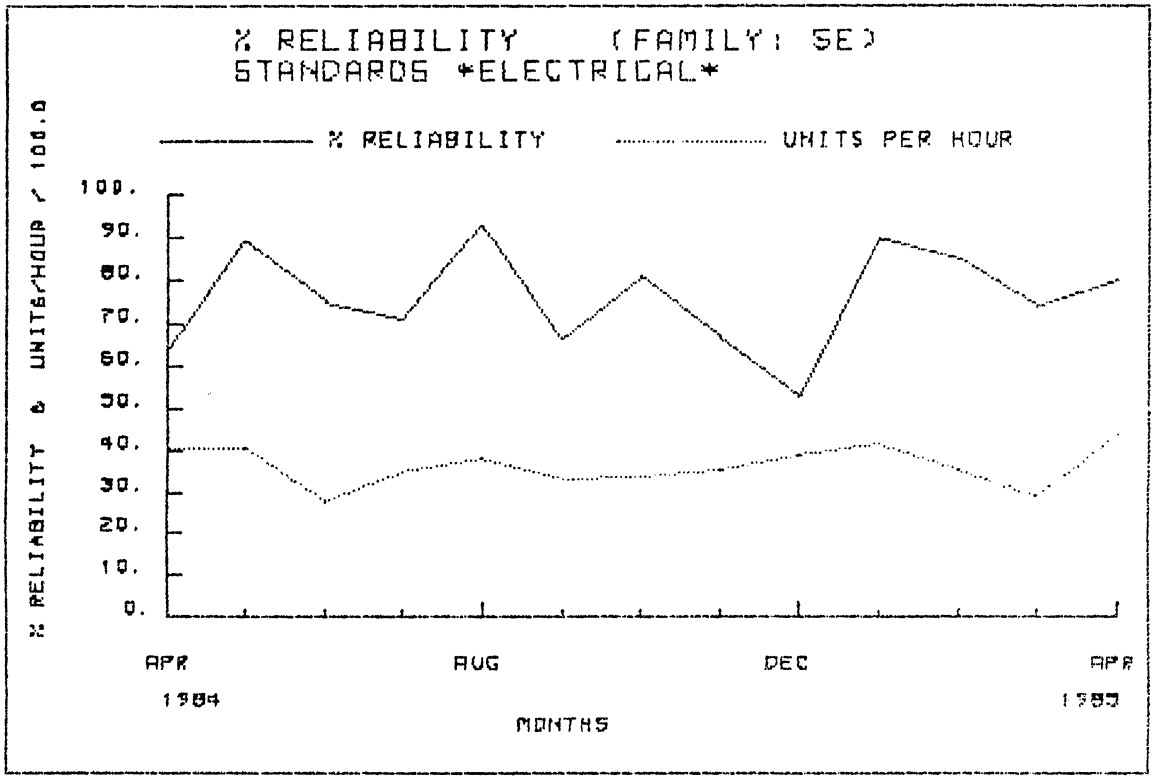


Figure 10. Family Reliability Plot

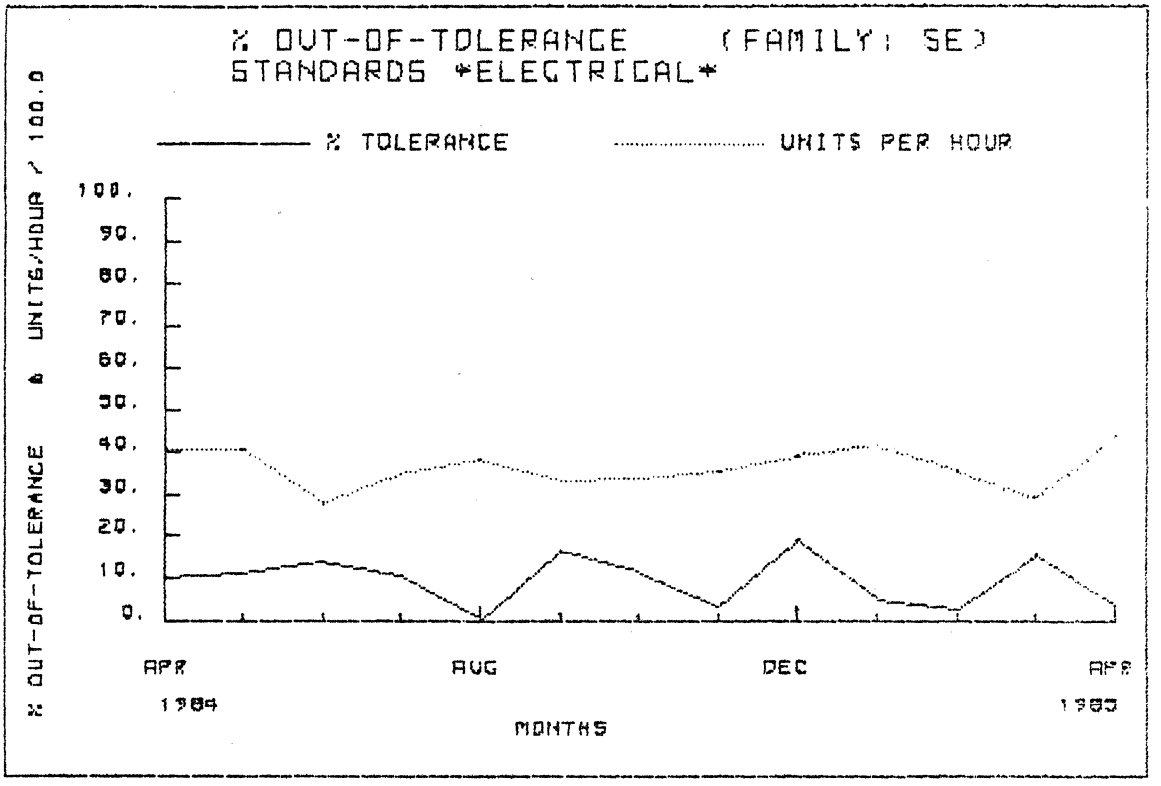


Figure 11. Family Out-of-Tolerance Plot

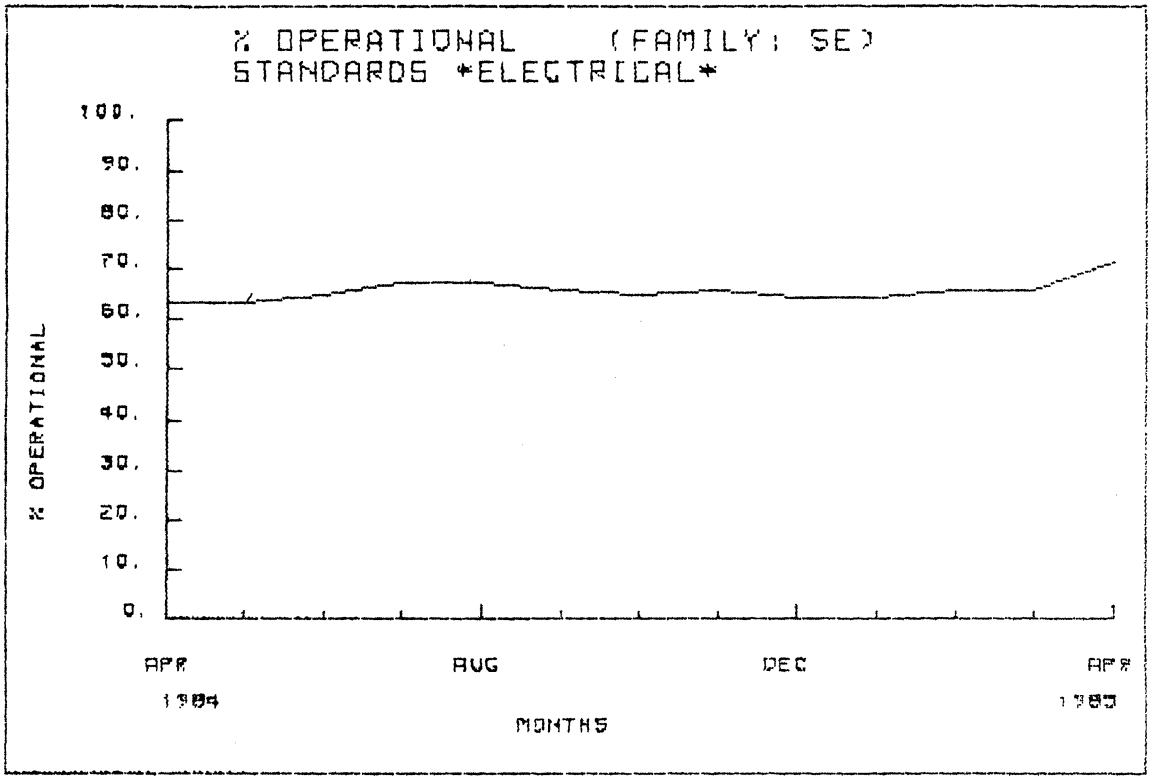


Figure 12. Family Operational Level plot

How To Buy Custom Software

Paul W. Miller
Corporate Computer Systems, Inc.
Custom Systems Group
33 West Main Street
Holmdel, New Jersey 07733

For the last 9 years CCS has been working with clients developing custom software packages to address a spectrum of needs. Our assignments have ranged from space satellite control to insulin needle test systems. Even with this diversity, one general theme has run through our interactions with customers. Most people don't know anything about the initial phases of custom software purchasing.

We discovered that during early contact, customers were most interested in the ultimate price for development. Unfortunately, trying to "just get prices" is the wrong approach to purchasing sophisticated custom software.

In the following, I will share with you some personal insights gained from fighting in the trenches of custom software development. I hope that this collection of observations, rules and war stories will help those of you who are embarking on what can be a very trying time -- the procurement of a custom software package.

Well begun is half done

The most important stage is of a custom software development effort is the beginning. Decisions made there can and do have a lasting effect on the tone of the entire project. Initial contacts with vendors, basic planning and contract negotiations are the foundations on which the entire project will be built. The foundations must be sound if the project is to be successful.

What are the stakes?

What is at stake when you begin development of a custom software package? At first thought you might be tempted to say the only risk you are exposing yourself to is the money you are going to spend on the development. After all, if the project fails that's all you can loose -- right? Wrong. Most of us work for other people. If you propose a project, sell it to your superiors, get funding and spend corporate time and treasure developing only to meet with failure, what is the loss? It takes a big chunk out of your career potential to say the least. Unless you can get yourself promoted before the final delivery, you have a very real

personal interest in the success of the project. If the project succeeds, you'll be a hero. But if it fails, you fail. Those are high stakes!

How do projects fail?

There are probably as many failure mechanisms as there are projects. There are some general trends though. I like to think that there are four basic class of failure:

1. It never got off the ground.

   This is the most obvious. For many different reasons some projects die before they are really born. The expense estimates were way off, the development fell far far behind acceptable limits or there were impossible (unanticipated) technical problems. Unfortunately, by the time you recognize it won't fly, tremendous resources have been expended. In retrospect, you always understand what went wrong -- too late.

2. Time bomb

   This is an insidious failure mode. The system is in, it passed the acceptance test, and is running. Suddenly, for no apparent reason, the system bellies up. Perhaps it only happens once or twice a week, but when it happens data is lost, work must be redone. Where is your support? How does the developer react? What does your boss think? We found that realtime systems have a nasty habit of displaying this symptom. The test loads just didn't have that "one" interesting case....

3. Great, but will they use it?

   One of the CCS systems I am most proud of was built right to specification, tested within an inch of its life and documented from the inside out, but it failed. The reason was that the people who wrote the requirements were from a different division from those who were to use the system. The users balked. The system sat. The project failed. From now on we require meetings with users as well as administrators.

4. Old age.

   When you think a system is done, it's not. Systems require care and tending and updates and changes. We have a client who worked with another custom house before us. The system they had installed was very old, unattended, and as a result had to be replaced entirely. Ongoing support and maintenance is important to a system's success.

Of course the cause is not hopeless. There are some steps that can be taken to save a project; to insure that it will be successful.

What is the real problem?

In order to solve a problem, you have to identify the problem. CCS was asked by a small surveying company to help computerize their collection process. After some investigation, we discovered that the company wasn't billing customers for jobs under $200.00. They didn't have a collection problem; they had an invoicing problem.

If we had gone by the initial description we might have come up with an elegant solution to the wrong problem. The first rule, then, is to try and really isolate the salient problem. This lets you establish the goals for the project. Try to make the solution result in a measurable benefit. "We want to automate the factory" is not really a good goal. "We want to increase productivity by 20%" is a good goal. Benefits might be financial, an improvement in quality or productivity or it might be better availability of information for important decision making.

What is it really worth to solve the problem?

Once the problem is isolated and the goals have been set, it is important to establish if the solution is worth what it will cost. If the benefit of the solution is financial, then the solution had better have a timely payback. If the benefit is availability of information, then the cost of that availability should be carefully considered. Do you really need that report? Computers are flashy for sure, but is one really needed for that particular function?

You may be surprised at the cost of developing custom solutions for computer problems. This surprise comes from two general sources. First, the cost of generally available software products (Visi-whatever and Calca-something) are extremely small relative to their actual development cost. This is because the costs are amortized over thousands of product sales. How many sales can your custom vendor use to amortize his costs? Right. Only one. Yours.

Second, developing software is a hard, dirty frustrating business. IBM's Little Tramp may be able to skate in with working software and Apple may be able to run the World Trade Center with a couple of quick keystrokes on a II/c, but we mortal developers are somewhat less deft at our trade. If Madison AvenQe's software doesn't work they just do a re-take; if ours doesn't work, we have to stay late at night, pay overtime, sweat blood and fix it.

Software, in general, is super labor intense. Custom systems are often mostly one of a kind (yes, we know about fourth generation languages, thank you) and as a result custom software is expensive.

What features should the system have?

Customers usually begin their interactions with us the same way "How much would it cost to ...." The question is easy to pose, but very hard to answer. In order to even come close to answering the "how much" question, the custom system developer must begin getting an idea of what the required features are -- all the required features.

No one would ask a builder "how much is a house?" We have to give him a hint on what we have in mind. Are there three bedrooms or four? Is there a swimming pool? Is it on a slab or is there a full basement. Software development is no different. We have to enumerate all the features the system is to have. It's useful to break the list into two parts. The "must have" list and the "wouldn't it be nice" list. These lists will enable you to evaluate different approaches to the solution. You must begin with a flexible attitude with respect to required features. Inflexibility will wind up adding substantial dollars to the final cost. Try to figure out what are truly required features.

Do you really need a custom solution?

As much as it pains me to say so, there are many cases where a custom solution is not warranted. Try and find an "off the shelf" product which addresses as many of your "must have" features as possible. Remember, you are sharing the development costs for the product with all of the other people who purchased it.

Now is the time to review your "must have" list. Is it really important that the company name appears in the upper left hand corner? Wouldn't the upper right be just as good? You might even consider changing the way you do things to accommodate an existing package. The cost savings could be large.

If one package won't do the trick, how about several? It may require some "glue programs" to develop the overall application, but these will be substantially less expensive than developing the system from scratch.

The rule to remember here is go custom if you really need to, but don't do it as a matter of course.

What is your people commitment?

Again, the Little Tramp is to blame here. Operating software

systems is not all sweetness and light. You have to assess your personnel needs in order to operate the system. Are you going to hire staff to maintain the sources or are you going to assume (hope?) that the vendor will be around to do it? What skill level of operator is going to be required to make the thing work? Sure he has to know how the application works, but will he also have to be a system manager?

Your approach may well be dictated by the people you have available. If you have people who are computer savvy, then perhaps you can piece together a system from off the self packages and a few home made glue programs. If you are not planning to have someone on staff who knows the answers, then make sure that your software vendor can support you in the long term. Also make sure that there will be adequate training and a friendly voice at the other end of the phone when you get stuck. At CCS we have 24 hour a day support -- some customers need it.

So you're going to do it custom, huh?

If, after, examining the alternatives you come to the conclusion that none of the products meet your list of needs, you should prepare to buy a custom solution.

The first surprise you will be met with is the complaint that your feature list is not complete. Custom systems builders who work on fixed cost are very conservative folks. They don't want to be half way into a job and discover that the customer really wanted it blue with red stripes instead of red with blue stripes. So, the first step will be to prepare a <u>functional requirements document</u>.

The functional requirements document will define precisely what the new system is supposed to do from the outside. I say from the outside because the functional requirements should not really include information about how the system will actually be built apart from where a particular implementation scheme is a requirement of the system (use IMAGE, or RTE, etc.)

If you develop the functional requirements yourself, be careful that they really say what you want them to say. Remember that this document is going to be a "treaty point" between you and your custom developer. The clearer the document is and the more precise the description of the requirements, the greater the chance you and the developer will complete the project and remain friends.

One really important aspect of the functional requirements often forgotten is capacity. How many, how much and how often are questions which should be answered. And the answers should be for tomorrow as well as today. The system, if successful, will most likely be asked to grow (you'll be getting promoted to handle the system's expanded responsibilities so growth is good). You will

be smart to consider the growth path right at the start.

One common mistake we have seen is undersizing the computing engine to save costs. Desk top computers are a prime example. Desk tops tend to be single user single program. A common growth path is to add multiple users and multi-tasking. This is very difficult with most desk tops. If you think this is the way the system will grow, it might be a good idea to buy a more general processor even if it means more up front money. The alternative may be to redevelop the system again for a different machine.

An alternative to developing the functional specifications yourself is to have a consulting company do this. If you take this route, make sure that you understand how many times you get to change your mind before the consultants become angry with you. You should set up an agreed upon scope for the development of your requirements.

Also, don't think that you have a lock on what the system is all about. Humility is important here. Remember your bows are taken when it all works -- who cares who contributes to the development? Make sure that the users, the secretaries, the president and the wash room attendant get a chance to tell you how the new system will impact their work. They may have important input. You should be in a listening roll here not in a defensive position. If someone insults your ideas, so what? You own the paper and pencil.

Sorry about this next section

Contrary to casual observation, the following is not an eye chart, it is a list of some topics which should appear in a functional requirements document. Not all topics should appear in all systems, but all of these topics should be <u>considered</u> for all systems.

A. Overview of the desired system
   1. Purpose of the system
   2. Current work flow
   3. Proposed system flow

B. Type of system needed
   1. Interactive - CRT immediate access
   2. Batch - overnight turnaround
   3. Combination - daily activity & summary reporting

C. Special system features
   1. Security requirements
   2. Interfaces with other systems & instruments
   3. Back-up, activity logging and recovery
   4. Archiving of data

D. Transactional activity
   1. CRT screen formats
   2. Report Layouts
   3. Plot formats
   4. Inquiries
   5. High & low volume of activity

E. Data definition
   1. Data items
   2. Key data items
   3. Collection of items into files
   4. File relationships
   5. Volume of data anticipated

F. Calculations & special algorithms

G. Hardware
   1. Number and types of interactive terminals
   2. Size of disc
   3. Throughput & response time required
   4. Telecommunications

So how much already?

Armed with a well written functional requirements document you are now ready to go out for bids. Make sure that you own the requirements document if you have had consultants develop it. That requirements document should be yours to do with as you please.

The functional requirements document is what should be given out to the potential bidders. We have some (non-governmental) customers who are really strict about bids. You must bid exactly what is in the requirements and they don't want to hear anything else. I think that this is a big mistake. Sure, it helps in the comparing apples to apples department, but it cuts you out of a lot of free engineering work. If a bidding company wants to suggest alternative approaches, I think they should be listened to -- especially if the suggestions are for free.

When you get your bids you should discuss what the engineering change procedure will be with each vendor. Undoubtedly there will be things which were left out of the specification. How will this problem be addressed? Also you will change your mind on things once you see pieces of the system operational. How will these changes be accommodated and at what cost?

Unless you are working with a trusted company with a track record, you should obtain several different bids. It is not uncommon to have a wide range between the lowest bid and the highest bid. Make sure that you understand why the low price is so low or why the high price is so high. You usually get what you pay for in software development. If you (inadvertently) trick a company into an artificially low bid through misunderstanding on either

their part or yours, success is doomed from the start (it won't get off the ground).

Who <u>are</u> those guys?

Before you award the contract, you should do some homework on the company you are going to pick. How long have they been in business? Who have they done work for? Were any projects similar to what you are going to try to do? Don't be shy. Ask questions and check out references. I have found that much of CCS business is repeat business. Assume that you are going to establish a long term relationship with your developer and make sure that they are worthy of the trust you are going to give them.

It sounds corny, but the construction of a software system is really a team effort. If an antagonistic relationship develops everyone suffers. A few checked facts can greatly contribute to the success of the project.

Who owns the software when the project is done?

This sounds like a silly question, but its not. You should discuss with your vendor who will own the software when the project is finished. Some vendors view custom systems projects as a golden opportunity to develop their software product line. Are you going to wind up grub-staking a vendor into a new product? Think about it, he could turn around and sell your software to your competition.

Another problem along this same line is the use of vendor proprietary packages in the development of your system. If the vendor gives you the sources of what you have paid for, but keeps key pieces because they are his proprietary products, what will happen to you if you must support yourself? An often workable solution to this problem is to escrow the sources of the vendor's product. The rules for the escrow must make allowances for his non-compliance with support commitments or his potential insolvency.

Don't be shy about asking vendors to sign non-disclosure agreements. You might ask for one before handing out functional requirements if you work in a highly competitive environment.

Where can I look it up?

The customer has a very basic and a very important decision to make. Will the customer provide the long term support or will the vendor? If you are going to support the system (support means enhancements, training, bug fixes and telephone calls late at night) then you are going to need the right type of documentation. Most documentation falls into two general categories: in-

ternal and external.

External documentation is what users typically receive. It tells you what to do before you hit the carriage return. This level of documentation is a must. If you don't get this you haven't got anything.

Internal documentation is what you will need to support the system. It should include a broad brush narrative of how the system works followed with a detailed explanation of individual modules. If possible, the internal documentation should be organized for reference. You will have a specific question which you want to ask. You don't want to have to read through reams of prose to find the answer. Look for indexes, tables of contents, etc.

Also an important part of internal documentation is annotated source listings. These are required, but don't let a vendor tell you that they <u>are</u> the internal documentation. You need some information on how the parts fit together. Don't just weigh the comments on source files, look at what is being said. If the variable X is set to 0 the comment should <u>not</u> be something like "set X to 0". Comments should be more descriptive.

Ask your prospective vendor for copies of documentation which he has provided to other customers as examples. Require that he provide at least this level to you. The cost of documentation may seem high and the temptation might be to cut some of it out. Don't. Documentation is you insurance policy against vanishing vendors.

Are we there yet, Daddy?

Get the vendor to provide best estimates for schedules. If time is of the essence on a project, be ready to pay more than if you are flexible about completion dates. Make a realistic assessment of what the target date means. If you are working to get something done for the next space shuttle, time might really be important. Other commitments could be less binding.

Your vendor is human too. He has other customers and employees which he must keep happy and busy. It is unreasonable to stall the bidding phase and eat up the vendor's development time by requiring a non-movable completion date. We keep running into customers who move slowly and then call up and say **NOW!** Even small vendors have inertia which must be taken into consideration.

Try to schedule working milestones. Progress payments are good for this. By taking this approach you get to see that you are really getting something for your money and the vendor gets some money for something. Always keep an interesting hold back pending completion of the work even if the vendor seems like an

honest guy.

One final aspect to completion is the acceptance test. The final test should be spelled out in the specification. If the details are not known, then the philosophy should be spelled out as clearly as possible. The acceptance test is your agreed sign off point with the vendor. This needs to be established for your protection as well as the vendor's. Projects sometimes have a way of going on forever because of poorly defined acceptance tests.

One year or 12,000 miles.

You can never get all of the bugs out of a major system, so a discussion of warranties is important. What does a warranty mean to your vendor? How long does it last? What will be the on going cost of fixing software which is out of warranty? Will the post-warranty support be available long term?

And now the hard part...

After you have successfully done all of the above it is time to award the contract and actually build the system. You can tell by the length of this section that I'm not going to tell you how to do that. Project management is an art in itself and worthy of many many more words. The only advise that I will offer here is to remember that you and the vendor should work together to insure the success of the project.

Try to understand at the beginning what roll you and your organization will play during actual development. If you have made commitments to do part of the development, such as review documents, make choices or develop test streams, make sure that these things are completed in a timely fashion. For some reason, people excuse their own slipped schedules while they are very unforgiving when others slip them. Don't give your vendor more to worry about by being late on what you owe him.

The literature is filled with horror stories about projects which never made it for one reason or another. Our experience has shown us that preparation done during the purchasing phase of a project can develop a sound foundation which greatly increases the likelihood of the project's overall success.

1006. TECHNIQUES FOR DEVELOPING DEVICE INDEPENDENT GRAPHICS SOFTWARE

Peter Neuhaus
Hewlett Packard Company
19447 Pruneridge
Cupertino CA 95014

Background

In the early 1970s, the computer graphics industry realized that it needed to standardize some of the methods used in developing graphics software. The resulting conventions made it possible to create graphics in one environment (computer) and transport them to another with a minimum of recoding. To date, only a few standards have been established but others are under investigation. The Graphics Kernal System (GKS) has been adopted by the International Standards Organization and is being used extensively throughout Europe while the Siggragh CORE system, proposed in 1979, has not gained much acceptance. The debate continues but GKS seem to be pulling ahead.

Regardless of whether or not one chooses to follow a strict standard, considerable improvements can be made in the writing of graphics software by following a few simple guidelines.

Frequently, companies plan to use only the specific graphics output devices that they already own, for example a HP7550 plotter or perhaps a non-HP graphics terminal. To support these devices, the specific instructions required by the devices would be scattered throughout the application program (see figure 1). The result would be very efficient but would necessitate excessive modifications if new or additional output devices were acquired at a later date.

The First Step

Device independence is nothing new to the professional programmer. Common functions such as cursor control are often modularized into separate subroutines (device drivers) that can be easily modified or replaced to accommodate new output devices that require different "escape sequences" for their proper operation. When it was necessary to drive more than one output device, a duplicate set of subroutines is written for each device (see figure 2). In addition, if more than one device might be used simultaneously, it is necessary for subroutines with identical functions to have different names, such as LINE1, for drawing a line on device 1, or LINE2 for device 2. At this level, device independence was still not achieved since the LINE1 and LINE2 calls must be embedded in the application program.

Step Two

By inserting another level between the application program and the device drivers, the interface between the application program and the outside world is standardized. If this new level, perhaps a commercially available GKS package, contains a function that allows the application program to select which output device should be used, if it is possible to remove the references to LINE1 and LINE2 and substitute a call to the new LINE function in the GKS package (see figure 3).

At this point, true device independence has been achieved since new devices can be supported without modifying the application program as long as someone writes a device driver for the new device. However, creating these new drivers can consume enormous amounts of programming effort because each device is unique in that it requires specific nonstandard "escape sequences" to perform a given task.

VDI - The Last Step

The graphics industry is attempting to standardize the hardware instructions required by graphic output devices through a concept called the Virtual Device Interface (now often called the Computer Graphics Interface). If all graphic devices understood the same commands, the need for device drivers would be eliminated (see figure 4). Essentially the device drivers would be implemented within the device's firmware. However, until the VDI concept becomes commonplace, it is necessary to employ the basic concepts of device independence when writing graphics applications. Several alternatives are possible.

Ways to be Independent

The most straightforward solution would be to obtain a graphics software library either from the computer manufacturer or from an independent third party. Such packages include any number of device drivers for the most popular graphic devices. The disadvantage to this solution becomes evident if it is necessary to change host computers at a later date. Even switching between computer lines offered by the same manufacturer can cause significant problems. Therefore, when shopping for this type of software product, it is important to investigate the possibility of moving the product between systems. Packages written in standard languages such as Fortran or Pascal help simplify portability. But even standard languages often don't port well.

The ability to move to another CPU may sound like something that wouldn't be done too often, but as desktop computers become as powerful as typical multi-user systems, many applications will be moved to smaller workstations. It's much like the user who feels he needs only 50 megabytes of disc storage, orders 100MB even though he knows it will never be needed, then runs out of disc space six months later. Applications and technologies change continuously. Investing the extra resources to implement a flexible solution often pays high dividends at a future date.

Sharing Graphics Data

Frequently graphic databases created on one system need to be processed on another. To address this need, a standard format for exchanging databases, called the International Graphics Exchange System (IGES), has been established and is currently supported by a number of graphics packages. A similar newer standard, the Virtual Device Metafile (VDM), performs much the same functions. By simply using the IGES or VDM device driver, an application can store the resulting image or object description onto a transportable media such a magnetic tape which can then be read by another IGES/VDM compatible system. Applications written in a device independent manner are able to utilize this useful feature.

Summary

The trade-offs involved in the decision to standardize the development of computer graphics software deal mainly with short term versus long term benefits. Projects that seem to be "one shot" programs may not appear to necessitate the features of device independence. But often the programs are modified and used again, possibly for another "one shot" application. In general, establishing standards or guidelines in a programming environment leads to increases in productivity. The slight performance degradation created by the overhead of a graphics subroutine library can be offset by the ever decreasing costs of computer hardware. Once standards have been implemented, applications can be developed faster since it becomes unnecessary to reinvent the wheel for each new project. In addition, program maintenance is simplified since each programmer understands the basic strategies used by his fellow graphics programmers. Overall, the need to be device independent will become increasingly important as the number and capabilities of systems and graphic devices expand.

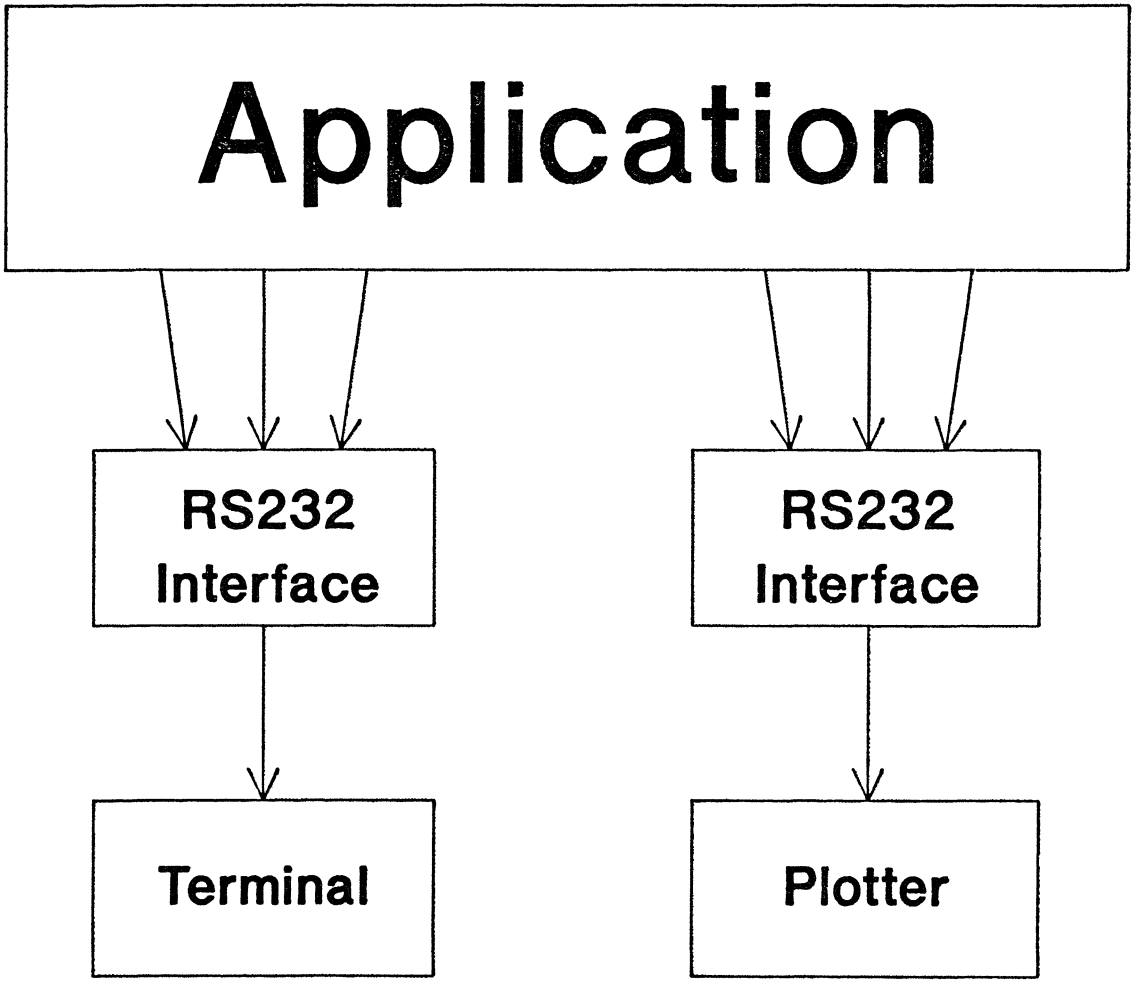

Application
RS232
Interface
RS232
Interface
Terminal
Plotter

Figure 1

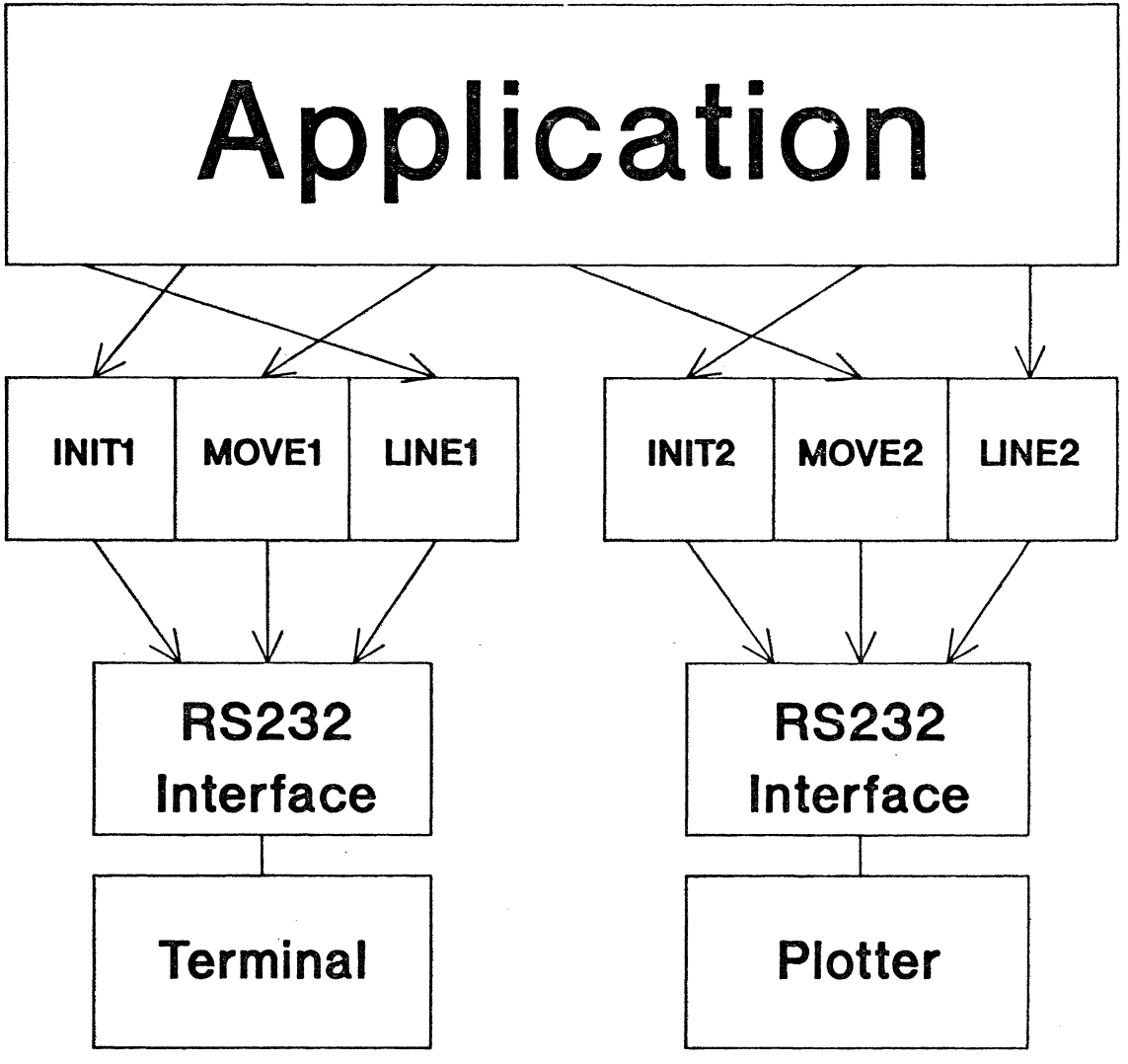

Application
INIT1
MOVE1
LINE1
INIT2
MOVE2
LINE2
RS232
Interface
RS232
Interface
Terminal
Plotter

Figure 2

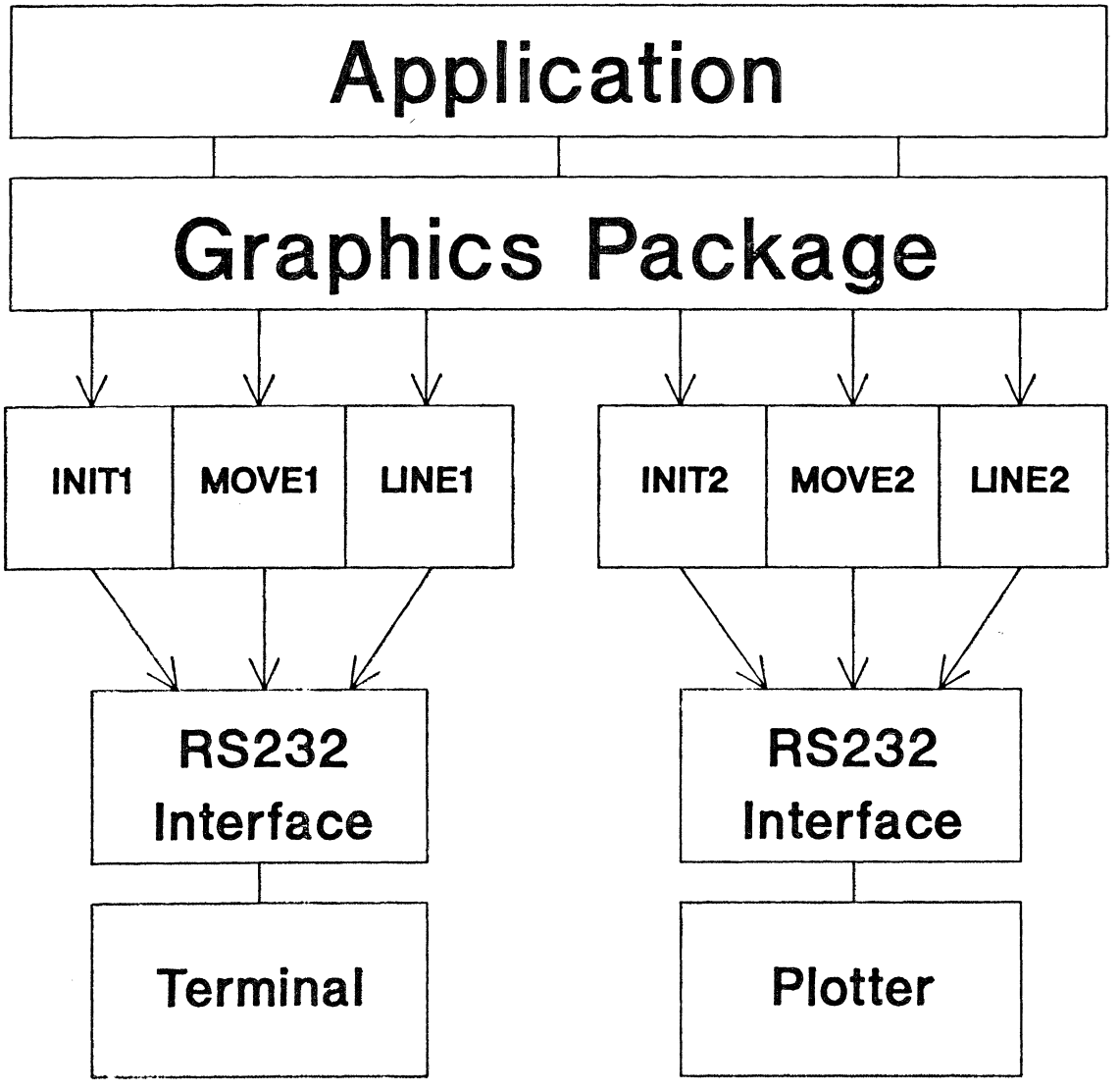

Application
Graphics Package
INIT1
MOVE1
LINE1
INIT2
MOVE2
LINE2
RS232 Interface
RS232 Interface
Terminal
Plotter

Figure 3

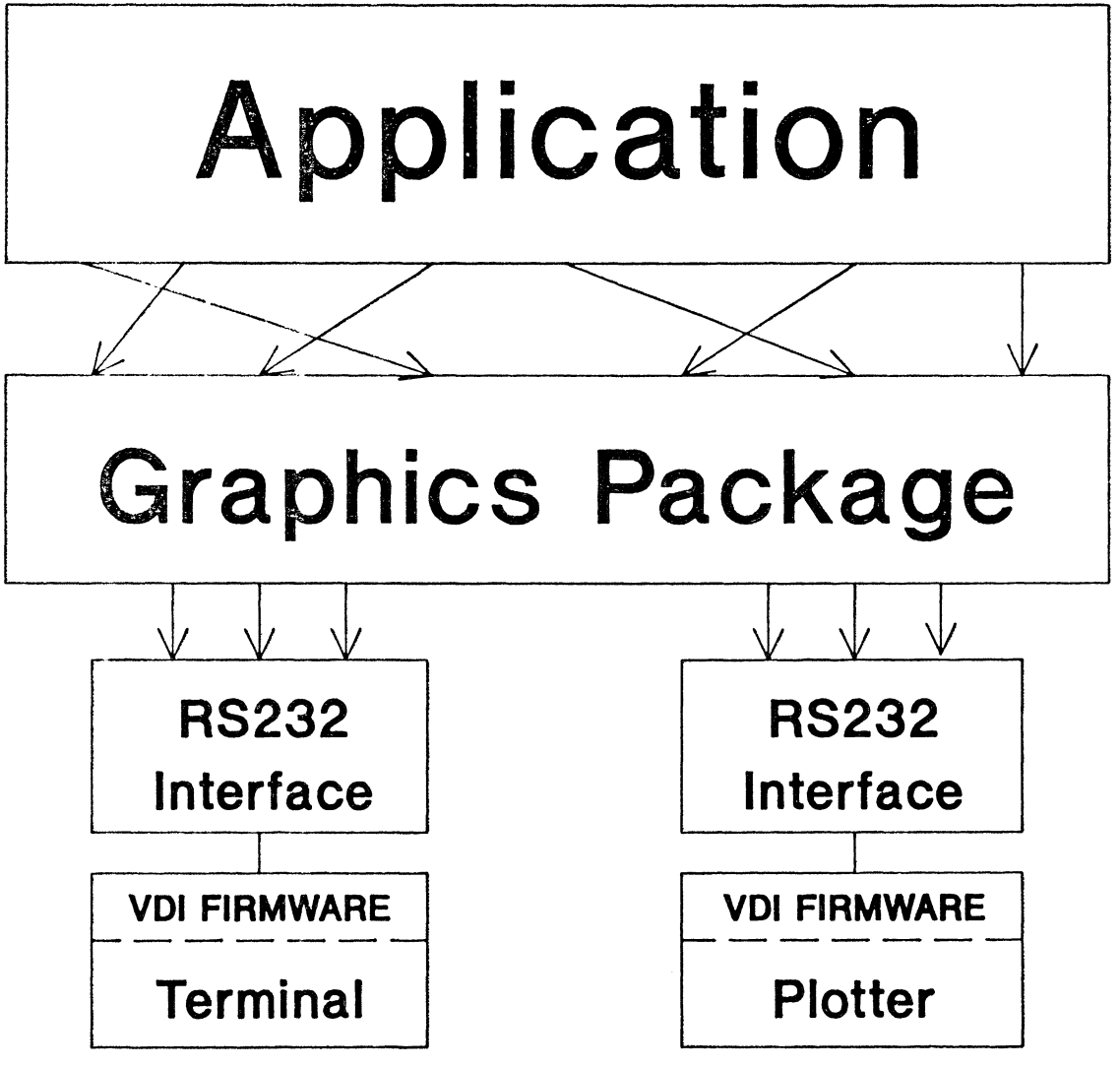

Application
Graphics Package
RS232
Interface
RS232
Interface
VDI FIRMWARE
Terminal
VDI FIRMWARE
Plotter

Figure 4

1007. TRANSIENT DATA ACQUISITION TECHNIQUES UNDER EDS

Steve Telford
Electronics Engineering Department
Lawrence Livermore National Laboratory
Livermore CA 94550

1. Introduction

The Experimental Diagnostic System (EDS) developed for the MARS project is used to support Atomic Vapor Laser Isotope Separation (AVLIS) experiments at Lawrence Livermore National Laboratory. The AVLIS process uses finely tuned frequencies of laser light to photoionize, or electrically charge, atoms of a particular isotope of uranium. The photoionized atoms are collected on charged plates. Reactor grade uranium can be produced by separating uranium-235 atoms from other naturally occuring isotopes. The purpose of developing the AVLIS process is to produce reactor fuel at a lower energy consumption and at lower capital and operating costs than other processes (gaseous diffusion or centrifugation). An experimental facility (Figure 1) was constructed at LLNL to study the AVLIS process. The project is divided into two areas: the laser facility and the separator facility. EDS was written to support the separator portion of the process.

The general philosophy behind EDS is to create a diagnostic to study specific aspects of the process. From this philosophy EDS has evolved into a general purpose diagnostic system which provides the user a means of acquiring large quantities of transient data, viewing the raw data as it is taken, and analyzing that data in real time. EDS also interfaces to our process control system, PMC/1000, and to our in house process modeling and analysis system. EDS consists of over 100 programs written primarily in FORTRAN and currently manages four diagnostics and eight users. The four diagnostics are:

- Vapor characterization by absorption spectroscopy
- Gas analysis by mass spectrometry
- Extractor Performance using transient recorders
- Process Laser Characterization

This paper will discuss the front end hardware used to do the data acquisition as well as the diagnostic control programs and data structures used to interface the front end hardware to the rest of EDS. Figure 2 is a general block diagram of the Experimental Diagnostic System.

EDS has been implemented on an HP/1000-A700 computer with 4 Mbytes of memory, 94 Mbytes of disc, and a 1600 BPI tape drive. Front end hardware is made up of CAMAC (IEEE Std. 583-1982) data acquisition and control modules and Tektronix digital oscilloscopes. Graphics output is made available on HP 26xx terminals and Raster Technologies color graphics systems.

2. EDS Design Elements

In order to develop a diagnostic system that we could readily adapt to the changing requirements of a very dynamic experiment, the following principle

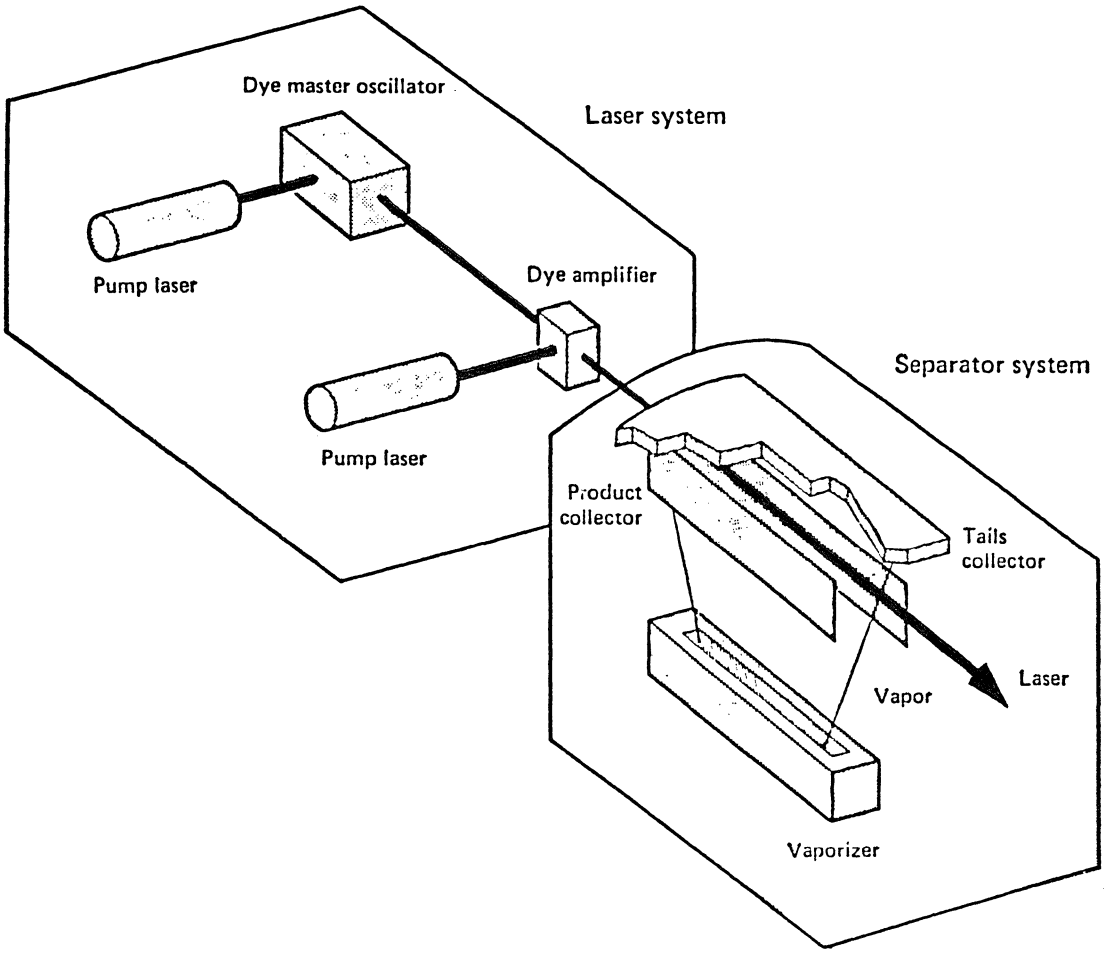


Figure 1. Atomic Vapor Laser Isotope Separation Diagram.

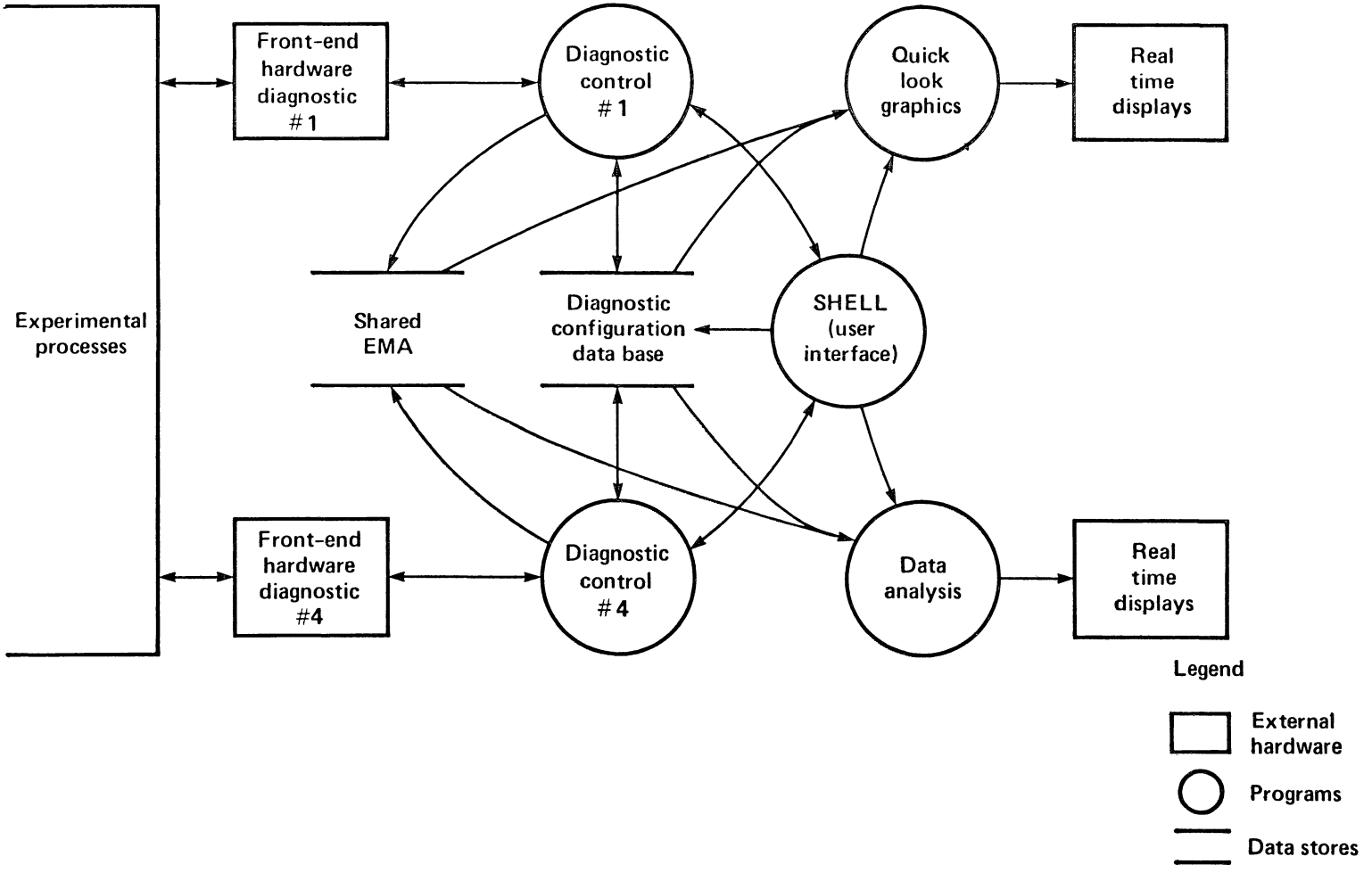


Fig. 2. EDS configuration.

design elements were identified:

- The front end hardware configuration had to be very flexible
- A method of logically connecting that hardware to the experiment was required (configuration data base)
- Large quantities of data had to be shared among many programs (data acquisition, graphics, analysis)
- Codes should be table driven
- Need a fast system
- Need a standard user interface
- Need a general graphics package

Figure 2 is a general overview of EDS showing all the major components of the system.

2.1 Front End Hardware Design Elements

In order to acquire large quantities of transient data the front end hardware must be capable of digitizing and storing that data without intervention or control from the computer. In the EDS system, control of front end hardware consists of set up, triggering, and unloading of the data. When and how often the data is sampled is controlled by the front end hardware after that hardware has been set up by the EDS system. All front end hardware used in EDS has local storage capabilities. By using front end hardware with local storage the communication to that equipment can be limited to small blocks (< 100 bytes) of programmatic I/O for set up and control, and DMA I/O for all large blocks of data (>100 bytes).

By using front end hardware with these characteristics the constraints on the CPU I/O performance can be relaxed. The DMA speed between the computer and front end storage device must be capable of moving all data acquired by the front end hardware very quickly. In the EDS system we have found that an end to end transfer speed of 50 Kbytes per second is the minimum we can tolerate. From our experience we have found that the instrument we are using for a particular diagnostic is the limiting factor on the data transfer speed, not the CPU. The GPIB interface (IEEE Std. 488-1978) offered by HP, for the A-series computers is well within our performance constraints and is a reliable means of interfacing front end equipment. All four of the diagnostics that we use in the EDS system interface to the front end hardware via HPIB.

2.2 Configuration Data Base Design Elements

The front end hardware used for data acquisition and diagnostic control is connected to the process being monitored through a configuration data base. The information in this data base provides a complete history of the front end hardware configuration for a particular experiment. The elements of this data base include:

- Which channels are active
- Engineering unit conversion algorithms and coefficients for all channels
- Short and long labels for each channel
- Relationships between the various data channels

- Where the data from each diagnostic event is stored
- The state of the front end hardware for each event

The data base not only holds the current diagnostic configuration, but also the configuration of the hardware for the duration of the experiment. This feature is necessary because often times what we are trying to accomplish changes in the middle of an experiment and therefore the diagnostic configuration must also change to meet the new goals. To meet these design criteria, a fast and reliable data base management system was needed. A third party data base management product, Berkeley Software System Database, written explicitly for HP 1000's, is used for all EDS data bases. This data base is used to keep the entire diagnostic configuration for an experiment. To ensure the best possible performance for EDS, key parameters from the data base are kept in Extended Memory Access (EMA) and only updated when the data base is changed. With this scheme we have been able to make EDS a very fast data driven system.

2.3 Shared EMA Design Elements

The shared EMA in our system is designed to make both raw and analyzed data from each diagnostic available to the entire EDS system. Because of the way HP has implemented EMA, only one shared EMA partition per program, all EMA for EDS had to be lumped together. Currently our EMA partition is just under 800 pages(1024 words = 1 page). For each diagnostic the raw data is read directly into EMA via the VMAIO call to the front end hardware. Once the data has been read in by the respective diagnostic control program it is immediately written to disc by the diagnostic control program. Here the data is read out of EMA into local program memory, identifying information is attached, and then written to disc. EMA provides a means of handling large quantities of data on the HP machines but has the disadvantages of 1) only one EMA partition per program, 2) no shared VMA, and 3) access time for a variable in EMA is triple that for a variable in local memory.

Another design philosophy of EDS required that the data acquisition, quick-look graphics, and real-time analysis portions of EDS be as independent as possible. In order to accomplish this the graphics and analysis phases of EDS always work from their own copies of the data. Once data is put into EMA by a diagnostic control program copies of that data are made by each program that wishes to operate on that data. This design allows each process to operate asynchronously from all other processes in the system. This also ensures that the performance of one process does not affect the performance of other processes and that the best possible system performance can be attained. The disadvantage of this method is that large amounts of memory are required to store multiple copies of all the data in the system (~ 800 pages).

3. Front End Hardware Configuration

On the MARS project we are using two general purpose hardware configurations for the four diagnostics in EDS. One configuration is based on a set of CAMAC modules and the other is based on high speed digital oscilloscopes.

3.1 CAMAC Front End Hardware

Figure 3 is a block diagram of the CAMAC hardware we use to interface three of

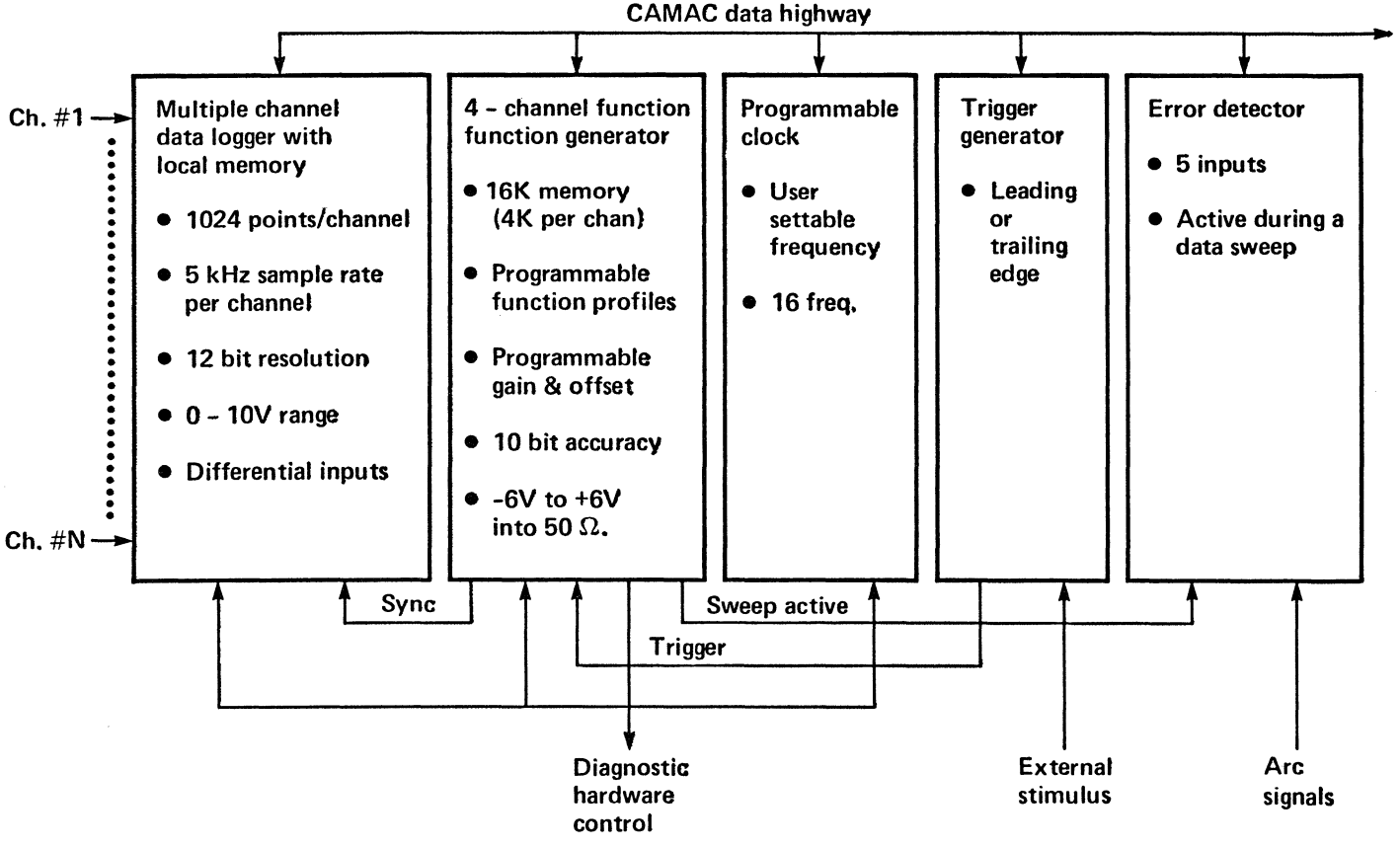


Fig. 3. CAMAC hardware configuration.

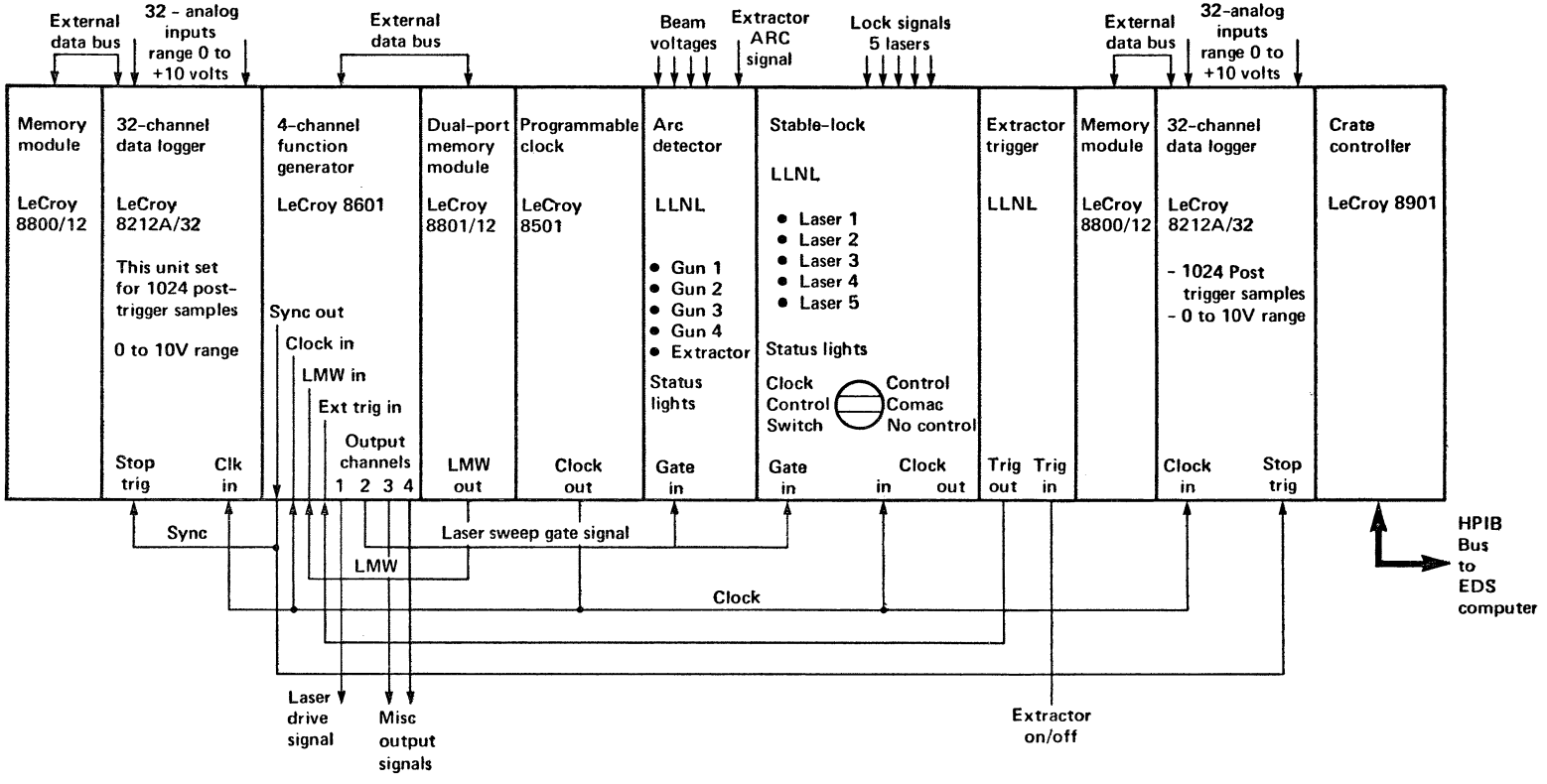


Fig. 4. Vapor characterization front end hardware.

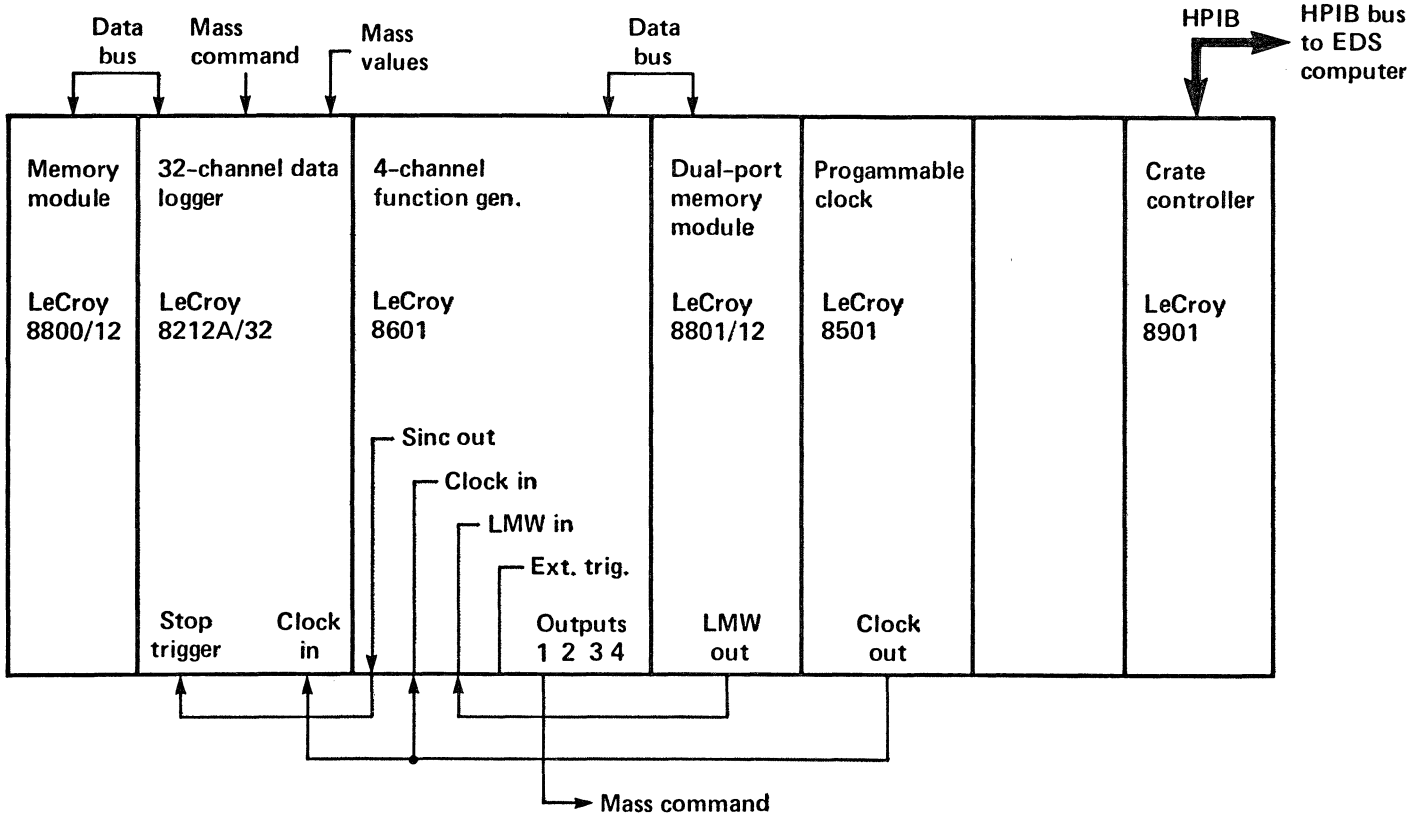


Fig. 5 Gas analysis front end hardware.

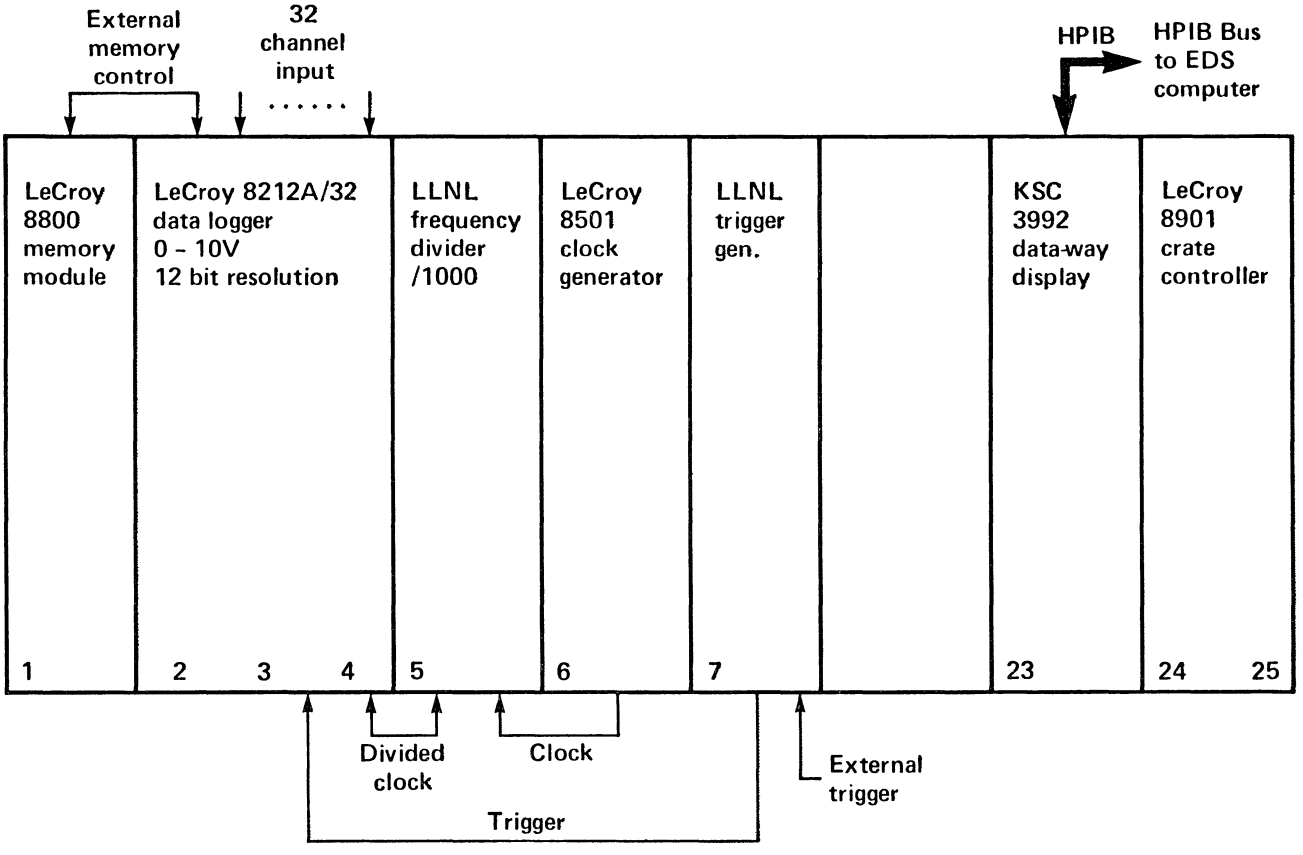


Fig. 6. Hardware configuration for process laser characterization.

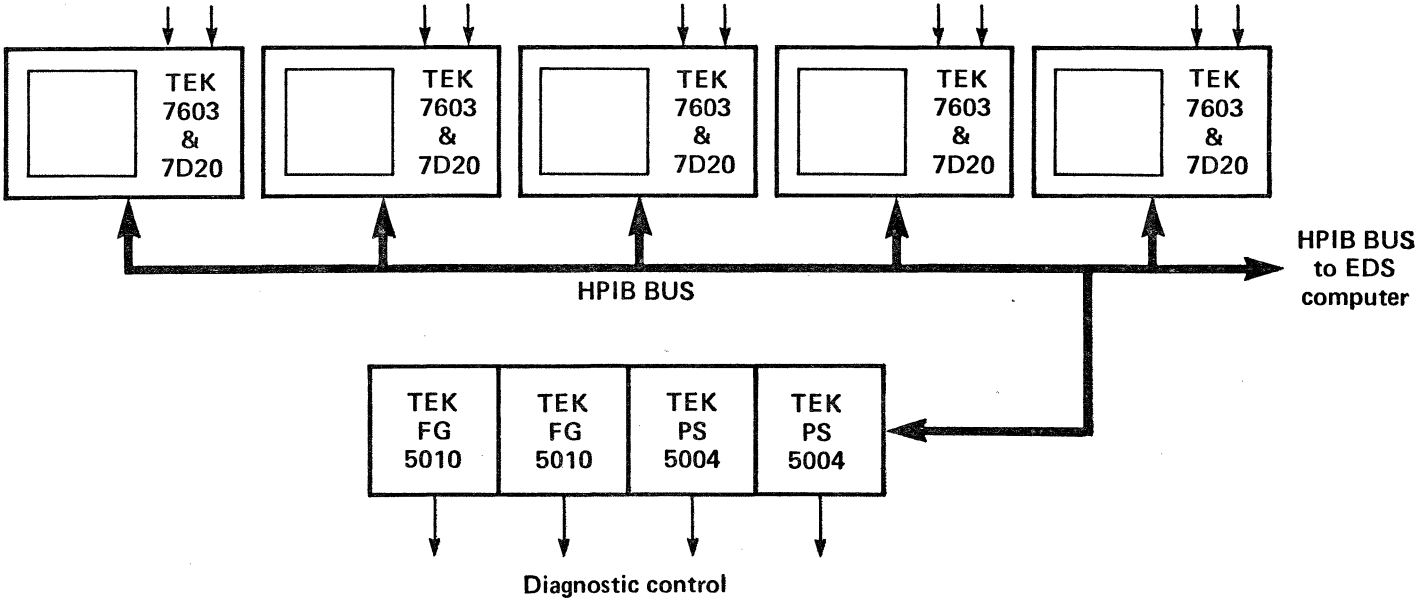


Fig. 7. Extractor performance diagnostic front end hardware.

our diagnostics. Figures 4, 5, & 6 are the detailed drawings of how the CAMAC equipment is configured for the Vapor, Gas Analysis, and Process Laser Characterization diagnostics. A four channel function generator is used to control any hardware associated with the particular diagnostic. The 32 channel data loggers are used to monitor all the signals associated with the diagnostic. As many 32 channel data loggers as necessary can be used for each configuration. Each data logger is capable of taking 5000 samples per second per channel and up to 1024 samples per channel can be held in local memory at each data logger. We have found this configuration to be very versatile. In order to pre-filter the data acquired by EDS we developed a general purpose error detection module that can be used to eliminate bad data before it is read into the computer. This module is active only when the front end hardware is acquiring data. At the end of a data scan, before the data is read into the computer, the status of the error detector is checked and if a problem occurred the front end hardware is immediately retriggered. Another module developed at Livermore is the trigger generator. This module allows us to trigger a data sweep in one of three ways: 1) operator command from the keyboard, 2) rising edge of an external stimulus, and 3) falling edge of an external stimulus. As shown in Figures 3 through 6 all the components described above are tied together through a common clock. This hardware configuration has the characteristics required to do a wide range of medium speed transient data acquisition: 1) Experimental hardware can be controlled by the diagnostic, 2) data acquisition equipment is able to acquire multiple channels of vector data simultaneously, and 3) the experimental configuration can be linked to this hardware via the configuration data base within EDS.

3.2 Digital Oscilloscopes

The second type of general purpose front end hardware that we use for diagnostics in EDS are digital oscilloscopes and programmable instrumentation built by Tektronix. This equipment allows us to control diagnostic hardware as well as acquire diagnostic data on multiple channels at a rate of 40Mhz per channel. Figure 5 is a typical hardware configuration using digital scopes as the main data acquisition device. As with the CAMAC equipment described above this hardware can be logically linked to the experiment through a real time data base. In addition to the increased sampling speed, these scopes offer the following advantages: 1) data can be previewed on the scope prior to being read in by the computer, 2) all scope settings are read and stored with the data, 3) these oscilloscopes have a wide input range, and 4) statistical functions can be performed on the data in the scope itself (i.e. signal averaging).

4. EDS Diagnostics

The following paragraphs will describe, in more detail, the four diagnostics that we have implemented under EDS on the MARS project. Each diagnostic consists of front end hardware tailored specifically for that diagnostic and a diagnostic control program which interfaces the front end hardware to the rest of EDS as well as to the real time data bases. In each case the real time data bases serve at least three basic functions: 1) they allow the experimenter to assign names and calibration factors to the input channels, 2) allow the experimenter to define relationships between the input signals, and 3) the data base is used to record exactly the state of the diagnostic each time data is acquired.

The diagnostic control programs associated with each diagnostic must meet the following guidelines in order to obtain the maximum performance and flexibility from each diagnostic: 1) CPU utilization by the control program must be kept to a minimum, 2) I/O must be as fast as possible due to the large quantities of data, 3) the user must be able to control all phases of the diagnostic from his terminal, 4) all pertinent information concerning the experiment must be saved in a data base as the experiment progresses, and 5) the control program must operate independent of the real time graphics and analysis. By meeting these criteria the diagnostic control programs are capable of controlling equipment and acquiring large amounts of data without putting undo strain on the CPU.

4.1 Vapor Characterization by Absorption Spectroscopy

Vapor diagnostics is designed to provide the user with the characteristics of the uranium vapor being created in the MARS facility. This diagnostic is based on absorption spectroscopy in a doppler broadened medium. Figure 8 is a block diagram of a typical configuration used to do absorption spectroscopy work on a uranium vapor source. Figure 4 is a diagram of the front end CAMAC hardware used to interface the absorption spectroscopy equipment to the EDS system. In order to support absorption spectroscopy experiments the front end hardware configuration must be able to accurately control CW dye lasers, acquire multiple channels of transient data simultaneously, and be logically connected to the experiment through a data base.

The four channel function generator is used to control the frequency of the dye lasers and the 64 channel data logger is used to monitor the intensity of the diagnostic laser light as well as other associated information about the experiment. The arc detector is used to monitor the status of the electron beam guns generating the vapor. The arc detector is only active during a laser sweep and can be queried at the end of each sweep before the data is read into the computer. If one of the guns had an arc during the sweep the data is discarded and the lasers are reswept for new data. The trigger generator is used to start a sweep from external stimulus. A laser sweep can be started by the operator from the keyboard, by the leading edge of a trigger pulse or from the falling edge. All of this equipment is then tied together through a common clock with a variable frequency. The hardware configuration described here has the characteristics required to do vapor diagnostics in real time: 1) experimental hardware can be controlled, 2) multiple channels of vector data can be acquired simultaneously, and 3) the experimental configuration is linked to this hardware via a data base.

The diagnostic control program, VCDL, is written to work explicitly with the front end hardware described above and meets the general guidelines for a diagnostic control program. This program allows the user to control all of the functions required to do vapor characterization, is responsible for archival of the data to disc, and keeps the real time data base current on each diagnostic laser sweep. Control of the diagnostic functions is communicated to VCDL through the EDS user interface, SHELL. Typical functions that the user can control thru SHELL are when to sweep the diagnostic laser, the profile of the laser sweep, the duration of the laser sweep, whether or not data is archived, and what conditions generate an automatic resweep of the lasers (arc mask).

When a laser sweep is initiated by the operator, VCDL sweeps the diagnostic

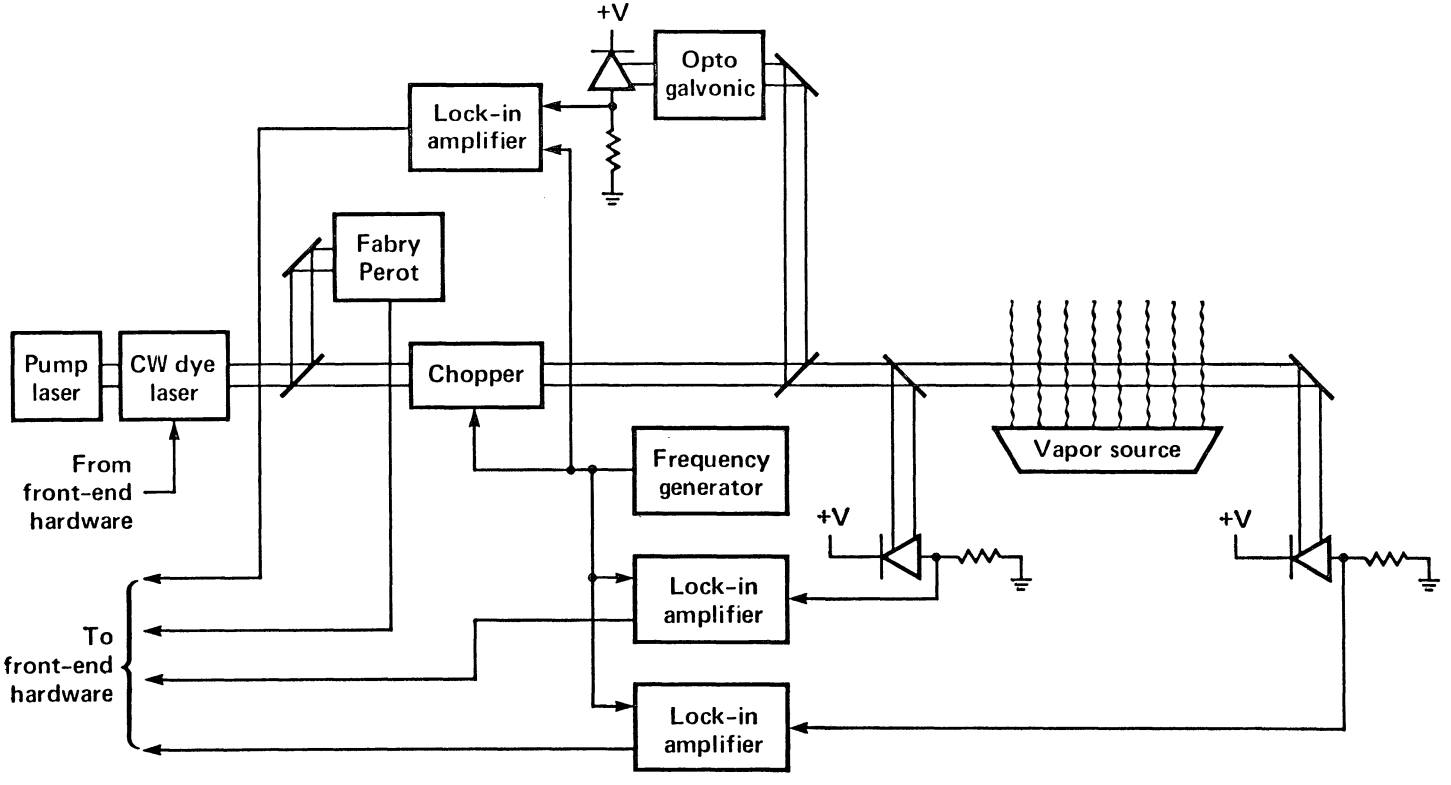


Fig. 8. Absorption spectroscopy hardware configuration.

lasers, waits for the sweep to complete, unloads the data loggers, archives the raw data to a disc file and places diagnostic status parameters into the real time data base for vapor diagnostics. When the data is read in by VCDL it is read into shared EMA partition so that the other pieces of EDS can get at the data. Once the data is read into shared EMA both quick-look graphics and real-time analysis make copies of the raw data to work from. Once the copies are completed the raw data can be written over with a new laser sweep. In this manner, as much independence as possible is maintained between the individual diagnostic and the graphics and analysis packages in EDS. This independence is maintained without sacrificing speed.

The combination of the front end hardware described above, the diagnostic control program, VCDL, and the real time data base create an environment which can be controlled by the user, changes with the experiment being performed, and does real-time vapor diagnostics.

4.2 Gas Analysis by Mass Spectrometry

The gas analysis diagnostic was designed to provide a real time indication of the relative amounts of different gases in our vacuum vessel. For this diagnostic the front end hardware and diagnostic control program must be able to control a UTI residual gas analyzer and monitor the output of that device. Input to the UTI consists of a DC voltage representing mass number and the output is a DC voltage representing the amount of gas sensed by the UTI. In order to get levels for a number of different gases the UTI is swept through mass numbers 0 to 50 using a function generator and mass values for those gases are monitored by data loggers.

The front end hardware used for this diagnostic is a subset of the hardware used for the vapor characterization diagnostic and is shown in Figure 5. The four channel function generator is used to drive the UTI through mass numbers 0 through 50 and two channels of the 32 channel data logger are used to monitor the drive signal sent to the UTI and the mass values returned. Both the function generator and the data logger are run from a common clock that has a programmable frequency. The hardware described here has the capabilities required to do real time residual gas analysis: 1) diagnostic hardware can be controlled, 2) multiple channels of vector data can be acquired simultaneously, and 3) the experimental configuration is logically connected to the front end hardware via a data base.

The diagnostic control program, MCDL, was written to work explicitly with the front end hardware described above and meet the general guidelines for a diagnostic control program. This program allows the user to control the diagnostic via the user interface, SHELL, is responsible for archiving the data to disc, and maintains current diagnostic status in a real-time data base. Typical functions that the user can control through SHELL are when to do an RGA sweep, the duration of the sweep, whether or not the data is archived, and whether or not to do quick look graphics and real time analysis with each sweep. If the user has chosen to do graphics and analysis it is the responsibility of the diagnostic control program to schedule these tasks. If these tasks are scheduled they are done so without wait so that the diagnostic can proceed independently from the graphics and analysis.

The combination of front end hardware described above, the diagnostic control

program, MCDL, quick-look graphics, real-time analysis, and a user interface combine to perform real-time gas analysis on the MARS experiment.

4.3 Process Laser Characterization

The process laser characterization diagnostic has been designed to monitor all the parameters in the MARS facility which determine the amount of separative work the facility is doing. This diagnostic is the simplest of the three described so far in that it is simply a slow speed data logger with real-time display and analysis of the data it is acquiring. Figure 6 is a diagram of the front end CAMAC hardware used to interface the signals of interest to the EDS system. Control of the diagnostic is done through the system wide user interface, SHELL, and plotting of the data is done through the quick-look graphics system. As with all other diagnostics in the system, status of the diagnostic is maintained in a real-time data base. In order to support the integrated separator diagnostics the system must be able to acquire multiple channels of slowly moving data and be logically connected to the front end hardware through a data base.

The hardware for this diagnostic includes only a 32 channel data logger and a programmable clock. As shown in Figure 6 the same clock is used here as for the other diagnostics but it is divided down by a factor of 1000 to better match the requirements.

The diagnostic control program for the integrated separator, ICDL, has many of the same capabilities as VCDL and MCDL described above. One additional command is available to the user with this diagnostic; he is able to set the rate the data is archived independently from the sample rate. The sampling rate is determined by how often one wishes the real-time plots to be updated and the archival rate determines how many points are plotted with each diagnostic scan as well as how many points are saved. The main function of the real-time data base for the integrated separator diagnostic is to attach name and conversion values to each signal coming into the data logger. The other function of the data base is to maintain the state of the diagnostic.

This front end hardware along with the diagnostic control program combine to create an environment where the user can monitor all the parameters associated with isotope separation in one location.

4.4 Extractor Performance Using Transient Recorders

This diagnostic is designed to provide the user with a means of monitoring very high speed transients yet make the data being taken available to the entire EDS community. Figure 7 is a diagram of the front end hardware we have selected to do this diagnostic. As one can see this diagnostic has been implemented mainly with Tektronix 7D20 digital oscilloscopes. A diagnostic control program has been written to control these devices and a data base designed to logically connect the front end instrumentation to the data being acquired.

Tektronix digital scopes were chosen for this diagnostic for a number of reasons: 1) they have a wide dynamic input range, 2) all scope settings can be read in along with the data, 3) data can be previewed before it is read in, 4) signal averaging can be done in the scopes to reduce random noise, and 5) the equipment is very easy to work with and interface to the computer. As this diagnostic

progresses control can be added by using Tektronix programmable instrumentation.

The diagnostic control program for this equipment is designed to simply pass user commands on to the front end equipment. This is the best way to communicate with this equipment because it already has a large (dt of English like commands that it understands. What the diagnostic control program does in addition to this is to allow the user to define an active set of scope channels and provide a small number of higher level commands which are strings of basic 7D20 commands. When a command is issued by the user, that command or string of commands is sent to all active scope channels as defined by the user. A diagnostic control data base is maintained and keeps track of the state of the diagnostic as well as where data is stored in the archive file for each data acquisition event. Once data has been read into the computer further viewing of it can be done through quick-look graphics.

5. I/O Performance of the Front End Equipment

I/O performance of the CAMAC front end equipment is governed by the crate controller used. On the MARS project we have been using the LeCroy 8901 GPIB crate controller. An evaluation of this has been reported to the HP 1000 users group in 1983 (UCRL-89129). Essentially this device will handle DMA I/O at rates up to 400 Kbytes per second. However in some instances the line level handshaking of the HPIB bus on the A-series computer is too fast for the 8901. In these instances the HPIB bus must be artificially delayed by placing a slower device on the bus. At Livermore, we have designed such a device into a CAMAC module that plugs into the same bus as the crate controller. This reduces the effective DMA speed to approximately 100Kbytes per second. In some cases where the CAMAC crate is placed more than 20 meters from the computer we have extended the HPIB bus with fiber optic bus extenders. The disadvantage of this method is that DMA speed is further reduced to 50Kbytes per second. But as I stated earlier we have found that an effective throughput rate of 50Kbtes per second is adequate. When we implemented the Tektronix digital oscilloscopes we found that we were able to communicate with them from the HP computer immediately. The transfer time for a single waveform of 1024 points and all associated parameters is approximately one second. We have never actually measured the DMA speed of these devices but have not found them to be a problem. One of the best things that can be said about the Tektronix scopes is that they worked as promised the first time.

6. Conclusions

EDS has been a successful integration of work done by many people over several years. It is unique at LLNL in the amount of data acquired, the graphics displayed, and the analysis done in real-time. The EDS design has withstood the test of time. While continually incorporating new commands, diagnostics, and capabilities, it has been run at regular intervals, one to three times a month. At the present time, EDS has been used in twenty (20) experiments with a total operating time in excess of one thousand (1000) hours.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

1008. HOW TO USE THE USERS

David E. Sullivan
ZONAR CORPORATION
2915 Hunter Mill Rd.
Oakton VA 22124

I suppose that in all areas of human endeavor there must always be a THEM and an US; continuation of the human race would be impossible without such distinctions.

But you'd think that once you'd climbed into bed with someone you could expect to be included in the resolution of common problems, and I for one have not always gotten that feeling during my six year marriage to Hewlett Packard.

Perhaps the too brief courtship and resultant shotgun wedding are partially to blame. I was brought in as a consultant on an HP 1000 based product installed by an OEM--then stayed behind to support the end user. Thus, no relationship was built with the local HP office.

But committed to HP we are, and after dealing with the HP support organizations over the years, experience has frequently made me feel that our THEM is not a large part of their US--at least in the area of system software problem solving.

Case I

Our first case study took place five years ago when we were developing complete software for a 30-user, interactive system. Reentrancy was considered necessary to avoid the RTE, copy-per- terminal approach which would require some 400 TASKS.

Since the JSB subroutine is inherently non-reentrant, the entire system was designed around the use of the (then) new JLY instruction.

Because of the reentrant environment, the on-line debugging facility available then was essentially unuseable; nonetheless, checkout was proceeding fairly well until we got to the "batch" programs which processed a large number of records on a continuous basis.

The more we tested the batch runs, the more "bugs" we found-- actually, we couldn't find the bugs, just the evidence of their occurrence (Remember DM TASK ABORTED?). And the more we tested the batch runs, the more the interactive TASKS didn't work either.

After six weeks of increasing frustration, we began to seriously seek help.

Our initial calls to the local HP office resulted in some sympathy; but since they hadn't heard from us in 14 months it took a lot of talking to convince them that at least we thought we had a problem.

The next step in the procedure, however, brought us to a complete halt. The HP office wanted to duplicate the problem on their HP 1000 configuration.

The configuration we were running on had 256 pages of memory, one 120 MB disc

and two 50 MB discs (full of data base), two tape drives and two sixteen line multiplexors with associated terminals; plus printers, a card reader, and console. In our experience, problems occurred with adequate frequency (four per day) only when everything was running.

Not only could their system not accommodate the data base, they didn't have thirty terminal operators available to exercise the system.

While asking the original OEM supplier of the equipment to work through his local HP office, we began to insert programmed traps around every problem area. The more traps we put in, the more non-trapped problems we discovered.

After two weeks, we began to get some trap information that allowed us to move the traps closer to the cause.

Productive use of the system was, of course, virtually impossible during this time. Every problem report called for analysis and "retrapping" before the function could be retried. Often, the problems resulted in data base damage which had to be corrected before the process could continue.

After four weeks of trap analysis, we felt we were getting close to discovering the cause and were sleeping in the computer room waiting for traps to go off. Meanwhile, both the local HP office and our OEM contact reconfirmed that no similar symptoms had been reported by any other HP user.

I had been begging continuously to talk to someone in the "lab" about the problem. It was my feeling that the problem would be recognized by someone in systems programming, once the details of our analysis were conveyed. But the HP field people kept saying that even they couldn't talk to the "lab", and we'd have to transport the problem to another site before they could do any more.

Finally the OEM presented another option. The HP rep from his area would come on site and take a look, if we agreed to pay the time and travel costs if it turned out not to be an HP problem--after more than two months, that sounded like a good deal.

I met with the HP rep and our OEM technical manager on a Thursday morning after spending all night in the computer room analyzing traps. I had a theory--the only thing that could explain the diversity of the problems we'd been having--but hadn't figured out a way to test it before the meeting.

All day was spent trying to bring the HP rep up to speed on the application system and the problem symptoms and analysis we'd accumulated. Although I didn't think he could be of immediate technical help, he was willing to accept the possibility of an HP problem; and he left that evening promising to use his contacts to research the problem through the "lab".

When he left, I went back to the computer room and wrote the following program:

```
START   NOP
        CLA
        CLB
LOOP    INA
```

```
                    JLY INCB
                    CPA B
                    JMP LOOP
                    HLT
INCB                INB
                    JPY 0
```

It ran less than 10 seconds before halting.

I took off Friday, after notifying the OEM of the transportable test; expecting to spend Monday in planning a work-around for the problem. But by Monday HP had installed a fix and the problem no longer existed.

Why were they able to install a fix so quickly? Of course--it was a known problem with a known fix!

Unfortunately, whoever had discovered the microcode error (which caused the JLY to NOP if interrupted during execution) decided that the fix should be provided only on user request.

The report of the problem/fix was made throughout HP in hardware release information, and its implications for software developers were apparently unrealized and therefore not communicated to the rest of the organization.

Although HP was obviously embarrassed, discussions with (mostly service) HP management did not result in any basic change in the way these problems would be handled in future. It did result in the one- time delivery of RTE source and documentation for our use--a penance which came to benefit HP at least as much as it did us, as you'll see in Case Two.

After finding the cause of the problem, it became evident that the "normal" problem resolution procedure had no hope of success.

1. The problem was not transportable because:

   a. it occurred in a complex environment with unique peripherals (in this case, 30 humans) which could not be emulated elsewhere, and

   b. it was hardware related, though as a design fault not component failure.

2. Checking known problem reports was useless, since:

   a. the problem was resolved within the hardware organization, and

   b. the nature of the problem defied intelligible description by the victim.

The case also suggests several truths which should have had an impact on the way such problems were handled in future:

1. One or more persons in the "lab" must have encountered and thus known of the problem and its impact on software. Only they would have recognized the symptoms, and only if communicated interactively and not up through the support channels.

2. The nature of the problem required an extremely high level of software expertise for its identification, and although it is sure to have occurred at other locations, credit for its eradication must go to a user working independent of HP support.

This initial experience with the field support structure also identified some philosophical differences between my beliefs, accumulated over a twenty year period, and those evidenced by HP management; such as:

1. Finished goods are not manufactured in a "lab"--so either the nomenclature is wrong or some other group is responsible for the software we use.

2. Good products, including software, require feedback from the consumer to be integrated into their design and production--preferably before this feedback is conveyed by a reduction in sales.

Case II

The second problem arose just as the last person in HP who might have remembered us from the JLY days was transferred or quit the company.

The symptoms were fairly straightforward--the computer hung in a tight loop, requiring a re-BOOT to restore operation. It happened once, and we blamed gremlins. It happened again, and we called service.

No amount of board swapping made it go away, and its frequency increased to eight times a week.

Since we hadn't changed system software in over six months, and since no one else had a problem like that, it appeared that once again it was OUR problem.

Weeks of analysis had gotten us to the point of blaming the double, sixteen-line multiplexor, and we then took an approach normally prohibited by HP policy--we looked at the source for the driver.

There it was! A sixteen millisecond window after RETURN was entered on a terminal when an additional input character would cause a tight loop. Obviously, some of our operators had learned how to shut down the system for early lunch, but of course they wouldn't tell us.

The most interesting part of this case was the subsequent process of communicating the fix to the "lab."

Although the local HP office never had quite understood the problem, it seems there was another site with similar symptoms; and the "lab" was interested in the fix.

A careful explanation communicated via the local systems manager made no sense by the time it got to California, and the "lab" decided to call us.

This couldn't be done easily, however, since the systems manager insisted that all conversations include him.

Thus I found myself standing in a phone booth in Massachusetts for half an hour while the Rockville manager tried to arrange a conference call with California.

Unfortunately, the initial, official fix didn't cover all aspects of the problem, and full resolution took two more conference calls and several more weeks.

This case illustrates some additional characteristics of the support environment:

1. Although the user is most interested in fixing a problem, effective use of the resources he is willing to invest requires some reasonable documentation and support--in this case source listings.

2. A production user needs first to restore productive operation. Typically, the standard HP support mechanism is only a last resort for accomplishing that objective. Thus, most frequently, users report problems first to help the HP community and second to try to stay "product line"--that is, to bring their fix into conformance with HP's.

3. The current store-and-forward path between the two individuals (one on-site and one in the "lab") who are competent to communicate makes both problem and fix reporting almost futile.

4. Finally, it raises the question; "who is the no- direct-contact policy intended to protect when the "lab" wants to talk to a user?"

Case III

The final case involves a printer that didn't work--at least not all the time. It never did work, and HP service finally gave up after trying everything they knew. They suggested the user work through his HP systems representative.

Unfortunately, this federal government organization didn't have a systems representative (OEM sale, again)--they just barely had Response Center Support.

So in spite of their expenditures for the original hardware and subsequent maintenance, there was no guarantee by HP that it would ever work on their system. And although they could buy guaranteed, two-hour response for hardware service, there's no such provision for a driver bug.

They could report the problem, which they did. And they could wait for new RTE releases which, thanks to the intercession of the OEM, were supplied through a truly concerned systems rep from Louisville on an emergency basis.

And they could run a new SYSGEN, at a cost of about five hours each, and find that the fix they'd installed was for some other printer problem.

Through these months and multiple SYSGENS, no one from HP ever wanted to look at conditions when the printer hung or at copies of the EQT dumps we had taken.

They weren't needed, of course, because they already had a site with THE PROBLEM and a fix was in process.

Each time we reported failure of a fix we were told "they" were already working on it.

OUR PROBLEM was ultimately fixed with another driver (installed magically without a SYSGEN) and the printer became fully operational 18 months after delivery.

I've been told that AMS would have made a difference; but the difference seems only to have included the ability to get early releases of the driver and allow installation without a full SYSGEN.

Both of these items should be available to any site with basic hardware and software maintenance, when a hardware component is not functional.

This case points up another defect of the problem resolution mechanism:

> There remains no method to assure that another, similar problem will not be assumed to be the same as one previously reported, until after the "fix" is installed and found not to work.
>
> This "take a number and wait" approach, although it may limit the total number of problem reports active at any one time, prevents delivery of complete problem information to the fixer, thereby dragging out the final, total solution.

A SOLUTION

I would first suggest that a description of the positive attributes of its USERS might precipitate better integration:

1. Users are at least as interested in resolving a problem as the assigned "lab" people, and may have a better test environment--especially for intermittent problems where getting it to occur while someone is watching is the trick.

2. For specific problem solving, users may be as competent as anyone in HP.

3. Once they've solved THEIR problem, most users will continue to work to eliminate the problem for others, on the assumption that others are doing the same for them.

4. Users don't pay their programmers and waste computer time to "play" with standard software. The only reason users might do non-standard, unsupported things is because they feel they have to in order to achieve or maintain production.

5. Production users are highly pragmatic--they'll use the least-cost

fix available and generally not assess blame until severely frustrated.

6. Users know they are stuck with HP, and work to feel good about it. Very few are masochists.

If HP can learn to take full advantage of their users, a number of significant benefits can result:

1. FREE MAINTENANCE - HP can use "free" user resources to improve their products and reduce their own maintenance costs.

2. FREE TRAINING - HP can allow the integration of ideas and information from highly qualified users into their design and programming groups.

3. FREE MARKETING - through a greater level of technical commitment to HP, its support, and its product line, by highly qualified industry professionals.

Unfortunately, the current support relationship is often an obstacle to HP's enjoying the full benefits of its user base, since it prevents utilization of some impressive resources.

THE REGISTERED EXPERT

It has been suggested that HP requires some protection from its users, lest it be inundated with trivia. What is needed, then, is a way to identify users with whom technical communications can be expected to be mutually beneficial. Since this depends on the individual person, a user category of "Registered Expert" is clearly required:

Qualification

Any individual may be recommended as a candidate for Registered Expert (RE), and upon confirmation of qualifications or expertise by an HP support person, the RE will be identified as expert in the appropriate, specific areas (e.g. RTE-IVB).

An RE must maintain his or her status by showing evidence of a minimum level of participation in the area(s) of expertise each year, upon annual renewal.

RE status may be withdrawn upon recommendation of any HP support person, thereby requiring re-qualification of the individual.

Privileges

The name, identity confirmation code, and areas of expertise will be maintained by HP on its RCS computer network for all RE's.

Initial contact within HP for RE's shall be at the same level as that

available to HP systems personnel.

RE's shall be able to file problem reports within their areas of expertise on a basis equal to that of HP systems personnel.

RE's shall be able to request call-back from increasingly higher levels of HP technical staff upon concurrence by the prior level; up to and including the "lab."

All conversations between RE's and HP personnel shall be considered as discussions between individuals, and neither shall be assumed to be speaking officially for his or her organization.

RE's shall be provided source files and available, internal documentation on items within their areas of expertise, at minimum cost.

RE's shall be provided SW service kits and other materials pertinent to their areas of expertise which are available to HP systems personnel.

None of the materials provided RE's under this plan shall be guaranteed or supported by HP. All shall be considered proprietary to HP if so designated, and use of such materials shall be limited to the cooperative, problem resolution efforts designated by the RE plan.

Upon recommendation of the appropriate HP systems personnel, an RE may accept primary or secondary responsibility for problem resolution. In such a case, local HP hardware and software resources will be made available to the same degree as if the effort were conducted by HP personnel.

Although all work by RE's shall be voluntary, and HP shall incur no obligation to pay for any such services; acknowledgment of the contribution made by RE's shall be made by HP in its Software Bulletins, or other appropriate documents.

Annual, informal RE conferences shall be held with HP "lab" and field personnel in appropriate expertise groups, for the purpose of trading information and ideas.

The formality of current HP software support channels has inhibited the construction of the "old boy's network" which exists in some other computer vendor environments. Thus, a formal approach to user integration has been proposed.

This plan will not solve all of the problems with the current support environment. It will, however, establish the necessary level of technical communication and understanding to allow their solution.

If such a program is considered by HP to be outside of the scope of their interest in the users, then some alternatives may be possible within INTEREX.

1. INTEREX problem reporting. With two publications each month it would seem there is room for user-generated problem reports. These reports would attempt to describe symptoms recognizable to someone else with

the problem; provide a "workaround;" and give contact information on the user submitting the report, so that additional information may be shared.

2. Provide pressure for problem resolution. INTEREX could maintain a list of the "TEN MOST WANTED" solutions, and use its power to hasten their development and distribution.

3. Maintain its own Registered Expert list of volunteers and a "hot line" to refer experts to one another.

The USERS remain an enormous, underutilized resource available to HP; and the RE program provides a feasible way to apply the best part of this talent to mutual advantage.

There must always be a THEM and US in dealings between customer and manufacturer; but commonality of interest (if not responsibility) provides the basis for a synergistic bond which can only benefit the entire HP community.

1009. AN IDEAL SYSTEMS DEVELOPMENT ENVIRONMENT EXPLOITING HP's FAMILY OF PROCESSORS

Dr. Hasan Sayani
ASTEC
9111 Edmonston Rd.
Greenbelt MD 20770

ABSTRACT

The development of powerful computers at reasonable prices has made possible the re-examination of the way resources are allocated to the Information Systems Development activity. While organizations employing CAD/CAM technology have routinely accepted that their engineers will have dedicated workstations at the $25,000-$30,000 level, the developers of Information Systems - administrative, real-time, and other - have typically not spent too much money beyond a "dumb" terminal for their analysts and designers.

This paper suggests a strategy of setting up a configuration of user work-stations (HP150's) to be used with an HP9000 at the center. Such an arrangement has also made possible the "front-ending" of complex development tools to make lesser experienced analysts and designers acceptably productive. The economics of the arrangement are shown to be quite acceptable, and other advantages for the analysts, designers, and their managers are discussed. An actual configuration is described along with scenarios of usage. Critical management issues necessary for the success of this arrangement are pointed out.

1010. EDS OPERATOR AND CONTROL SOFTWARE

Linda L. Ott
Lawrence Livermore National Laboratory
P.O. Box 808 L-441
Livermore, California 94550

1. INTRODUCTION

The AVLIS process uses finely tuned frequencies of laser light to photoionize, or electrically charge, atoms of a particular isotope of uranium. The photoionized atoms are collected on charged plates. Reactor grade uranium can be produced by separating uranium-235 atoms from other naturally occurring isotopes. The purpose of developing the AVLIS process is to produce reactor fuel at a lower energy consumption and at lower capital and operating costs than other processes (gaseous diffusion or centrifugation). An experimental facility (fig. 1) was constructed at LLNL to study laser isotope separation. The project is divided into two areas: the laser facility and the separator facility. EDS was written to support the separator portion of the process.

Two types of data were identified as being necessary to collect: the conventional, relatively slowly changing scalar values (temperatures, pressures, etc.) and large vector arrays of fast transient data. Many commercial systems are available to monitor the slowly changing data and the decision was made to use the HP product PMC/1000. To collect the highly specialized types of data, the data acquisition system (EDS) was developed in-house. EDS is functionally divided into three parts: data acquisition hardware and software, control and graphics software, and analysis software. This paper describes the second function: control and graphics.

The general philosophy behind EDS is to create a diagnostic to study specific aspects of the process and then, once it is sufficiently characterized, to replace the diagnostic with a "black box" to provide the condensed data to the plant process and control system. For example, during one vapor diagnostic sweep cycle, 64K words of data are reduced to approximately ten values (eg. temperatures, velocities, densities). The natural progression for an EDS diagnostic is to migrate from an interactive experiment to a passive instrument.

2. DESCRIPTION

The Separator Instrument and Control system is based on a network of five HP1000's:

Node 1:	Post Run Analysis	(F series)
Node 2:	PMC/1000	(A700)
Node 3:	EDS	(A700)
Node 4:	Feeder system	(A700)
Node 5:	Modeling	(A900)

EDS consists of over 100 programs written primarily in FORTRAN. EDS has evolved over a number of years from a one user, one diagnostic system to a multiple user, multiple diagnostic system. EDS currently manages four diagnostics and eight users.

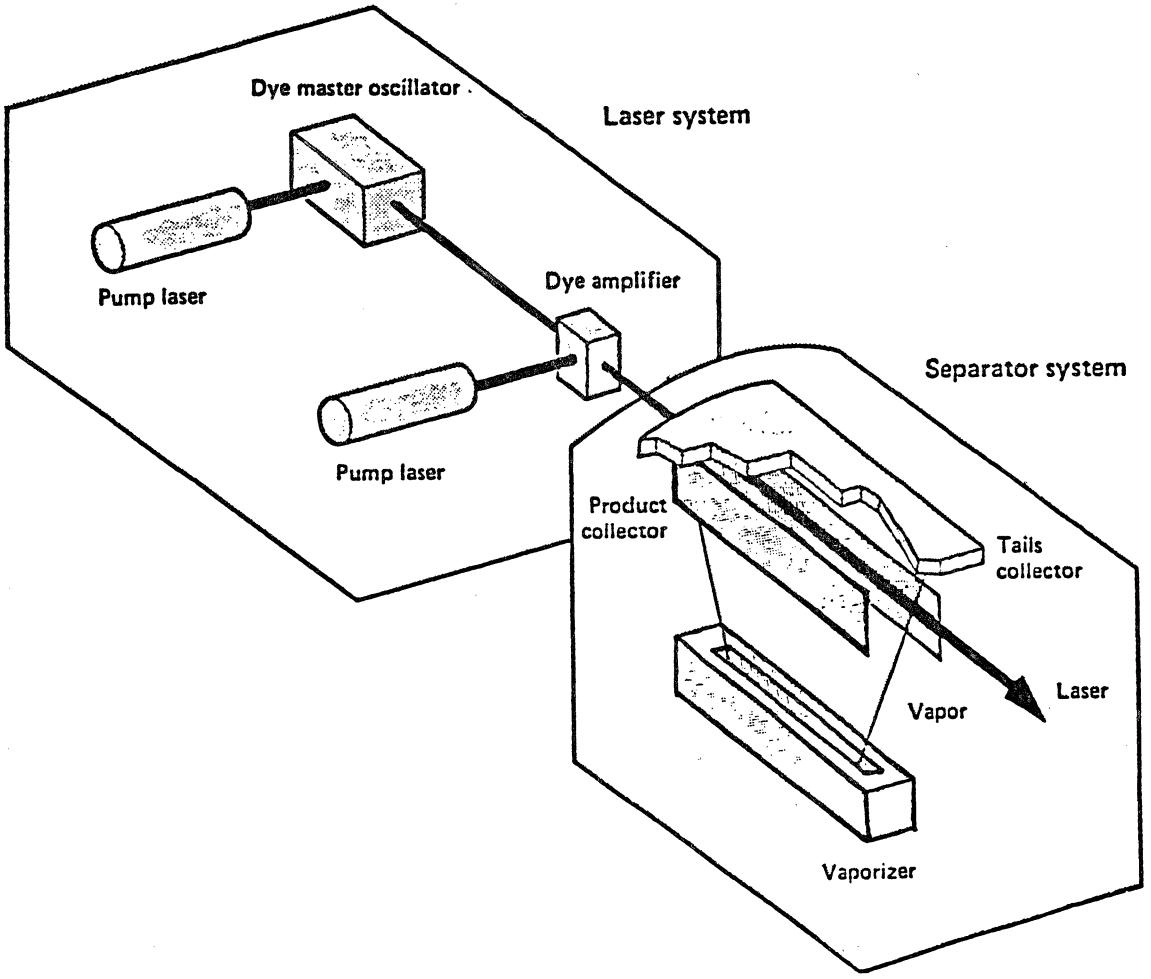


Figure 1. Atomic Vapor Laser Isotope Separation Diagram.

The four diagnostics are:

* Vapor characterization by absorption spectroscopy
  (64 channels, 1024 words per channel)
* Gas analysis by mass spectrometry
  (10 channels, 1024 words per channel)
* Extractor Performance using transient recorders
  (20 channels, 1024 words per channel)
* Process Laser Characterization
  (32 channels, 1024 words per channel)

A complete diagnostic cycle consists of data acquisition, display, analysis and transmission of key process parameters to the process monitoring/control system, PMC/1000, and to the process modelling system.

3. GENERAL REQUIREMENTS FOR EDS

The requirements for EDS are listed below:

* Acquire large quantities of transient data.
* Create a convenient user interface.
  - Provide means to configure workstation displays.
  - Provide simple commands to manipulate diagnostic.
* Provide guarantee that only one user is controlling a diagnostic at a given time.
* Design interface to incorporate new commands easily.
* Allow several levels of command execution to handle a range of user sophistication.
  - no interaction
  - detailed interaction
* Permit different modes of diagnostic operation.
  - Automatic Mode: Used when system is stable. Scan at specific intervals.
  - Manual Mode: Used during critical transition periods. User has complete control.
* Ensure that the first priority is data acquisition. Graphics and analysis are to be done as time allows.
* Provide immediate results of scan.
  - Graphical output should be displayed within ten seconds of the scan's completion.
  - Analytical capabilities should be user defined and modifiable during the run.
* Make data available for display at any workstation.
* Need ability to isolate or coordinate diagnostics.
* Need convenient means of adding new diagnostics.
* System needed immediately. The deadline was moved from 1987 to April 1985.

4. EDS DESIGN ELEMENTS

After studying the general requirements for EDS, the following principle design elements emerged:

* Need the ability to share large amounts of data among

many programs (data acquisition, graphics, analysis codes)

* Need to make codes data table driven.
  - Write codes to accept input from data bases.
  - Write interpretive codes that receive input from text files.
* Need to make data acquisition, graphics and analysis independent of each other. The system should be uncoupled.
* Need a fast system.
* Need a standard user interface.
* Need a general graphics package.

5. EMA DEFINITION

Shared extended memory area (EMA) was selected as the fastest means of sharing large data arrays among programs. Twelve named common blocks were created using a total of 778 pages (1024 words per page) of memory area. The common blocks are listed below:

* Vapor diagnostic raw data and parameters (65 pages)
* Mass spectrometer diagnostic raw data and parameters (33 pages)
* Extractor diagnostic raw data and parameters (21 pages)
* Integrated separator diagnostic raw data and parameters (33 pages)
* Vapor analysis work area (77 pages)
* Mass spectrometer analysis work area (9 pages)
* Extractor analysis work area (41 pages)
* Analysis matrix (165 pages)
* Graphics data area (202 pages)
* Graphics user tables (97 pages)
* Graphics viewport tables (9 pages)
* Graphics element tables (26 pages)

Each diagnostic has its own common area which contains the raw data and the parameters necessary to control the diagnostic. The raw data is duplicated in both the graphics and the analysis work common areas. The cost of keeping graphics and analysis independent of data acquisition and of each other is large memory usage.

6. EDS INTERFACE PROGRAM (SHELL)

The SHELL is the primary interface between the experimenter and the rest of the EDS software. The main purpose of the SHELL is to provide the user with a mechanism to programmatically control a diagnostic, analyze the data and graphically display the raw and reduced data. Basically, the SHELL schedules programs and passes commands to active programs. It isolates the user from the

HP operating system. All commands to EDS are executed through the SHELL. System log-on files are created for each diagnostic which set the working directory and run SHELL with the appropriate diagnostic name. The first prompt the user sees when he logs on to the EDS system is the SHELL prompt for his diagnostic.

Commands are grouped in the SHELL by category. Each diagnostic has its own category. Once a user has obtained ownership of a category, he has exclusive access to the commands in the category. This ensures that only one person is controlling a diagnostic. Commands within a category are two characters followed by parameters. If parameters are required but not entered, a menu is presented to prompt the user for input. The parameters are range checked and validated before passing on to the appropriate program. The SHELL is not connected with EMA.

All commands are contained in the SHELL database. When new commands are needed, they are entered into the data base. The SHELL program itself is not modified. The following elements define a command in the data base:

```
Category          (two characters)
Subcategory       (two characters)
Purpose           (forty characters of text)
Action selector   (two characters)
  Use class I/O
  Schedule program without wait, single copy per session
  Schedule program with wait, single copy per session
  Schedule program without wait, single copy per system
  Schedule program with wait, single copy per system
Name of program to schedule or receive class I/O message
Parameter requirements
Number of parameters associated with the command
Parameter type    (integer, real, ASCII)
Parameter definition
Parameter key words
Parameter default values
Parameter lower limits
Parameter upper limits
Forms library name
Menu number
```

The standard format for a message from the SHELL is an integer ASCII string. The string contains the SHELL data base manager name, the SHELL workstation, the command, and the parameters.

7. SHELL HELP LEVELS

Several levels of help are available. Commands within a category are grouped by function into subcategories. Two question marks (??) will list all the categories and subcategories in the SHELL. If a category is owned, the owner will also be listed. A question mark followed by a category (eg. ?VC) will list all the commands available in the category; a question mark followed by a subcategory (eg. ?V8) will list the commands in the subcategory with a short description of the command. A question mark followed by the command (eg. ?VCSW) will list a detailed explanation of the command. Users may select different levels of command execution from immediate execution when the command is entered

to stepping the user through each phase of command execution with a prompt.

8. WORKSTATION CONFIGURATION

Through the SHELL, the user configures the workstation's graphics device, typically either an HP2627 display terminal or a CONRAC monitor driven by a Raster Technologies Graphic Display Controller. A Versatec printer/plotter may be selected as the graphics display device although this is usually not done in the real-time mode. For the Raster Technologies device, there are generally two graphic planes defined with either four or eight viewports per plane. In this manner, the user has eight to sixteen graphs available at one time. Plots may be superimposed; line types, colors, and symbols are selectable. The user also has the capability of plotting one data set against another. The data may be reduced in a number of ways before being plotted by defining functions and coefficients in the data base. The in-house Device Independent Graphics Library (DIGLIB) has been very valuable in meeting the EDS graphic requirements.

9. TYPICAL DIAGNOSTIC PROCEDURE

Perhaps the best way of illustrating the operation of EDS is to describe a typical diagnostic run session. The vapor diagnostic will be used for the example. Be aware, that the other three diagnostics could be running on separate workstations at the same time. It would be possible for one user to "own" and control all four diagnostic at once. The basic sequence of events for a diagnostic is:

* Set up instrument(s)
* Get data
* Plot raw data or slightly transformed data
* Condense the vector data to scalar parameters
* Plot the reduced data
* Send the reduced data to other nodes

A typical procedure is outlined below:

1. Log-on to the computer and EDS.
   a. The user types VAPOR.
   b. The SHELL prompt (VC>) appears for the vapor diagnostic.
2. The user turns "ON" his diagnostic
   a. CAMAC instruments are initialized.
   b. Update EMA with the data base values (Fig. 2)
      * Define active channels
      * Define labels
      * Define engineering conversion algorithms and coefficients
      * Define analysis parameters
   c. Set up Class I/O communication link between the SHELL and the message handling program (VMESS).
3. The user selects the graphs to be displayed.
4. The user selects analysis options:
   * Don't do any analysis
   * Calculate reduced parameters
   * Calculate and plot reduced parameters
5. Initiate data acquisition.
   The user may select to do sweeps one at a time,

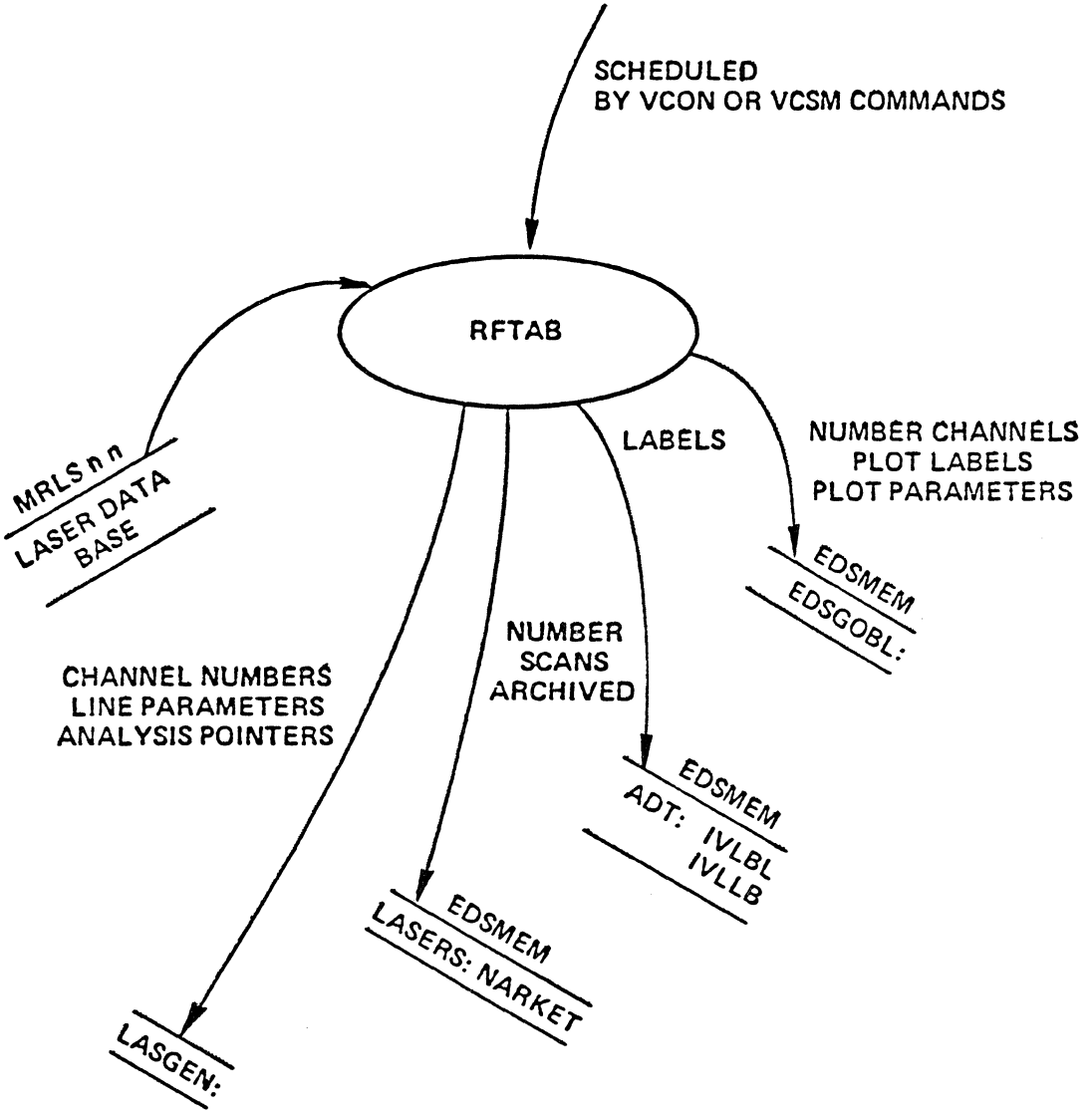


Figure 2. Update EMA with Data Base Values.

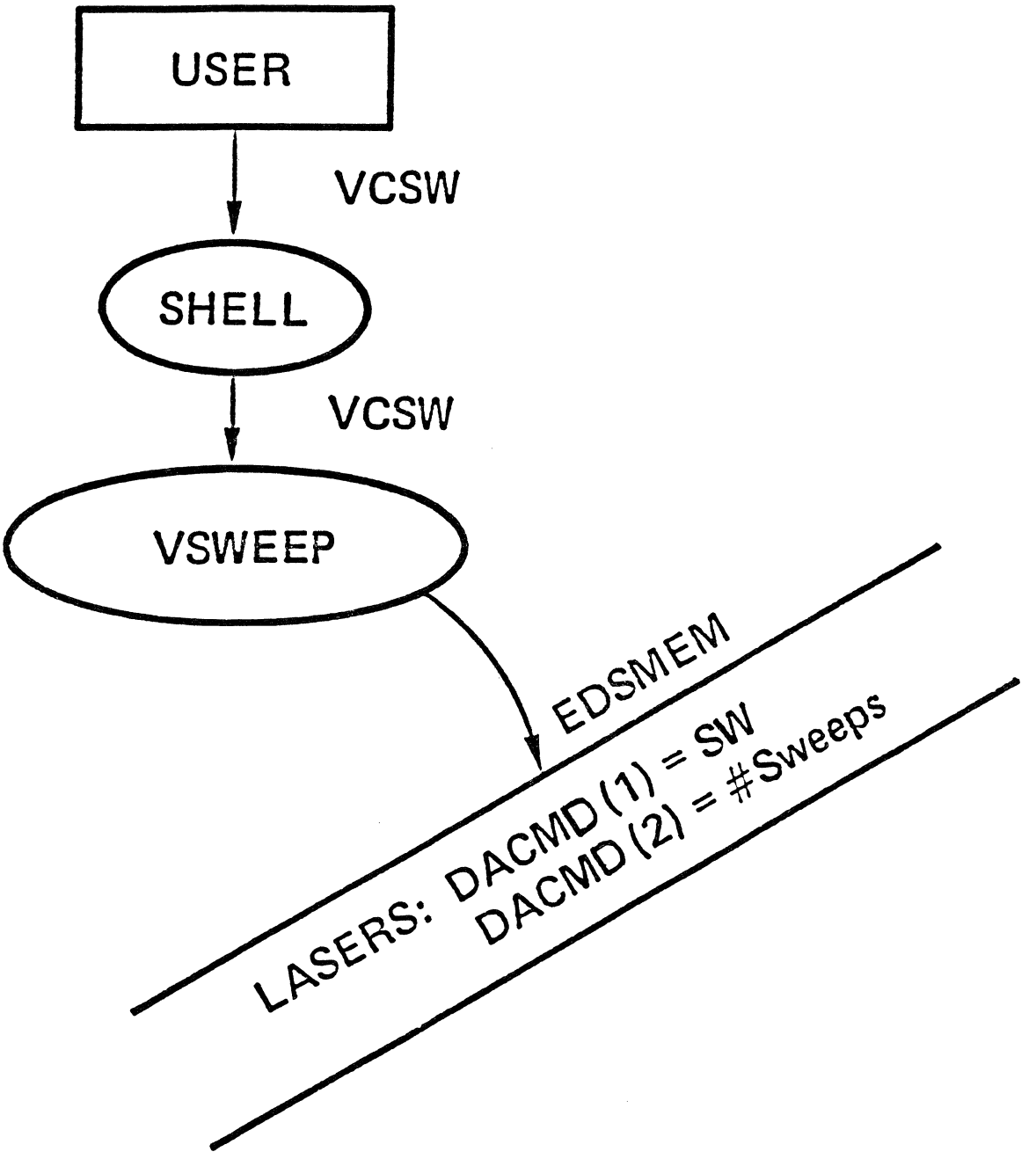


Figure 3. Execution of a Sweep Command.

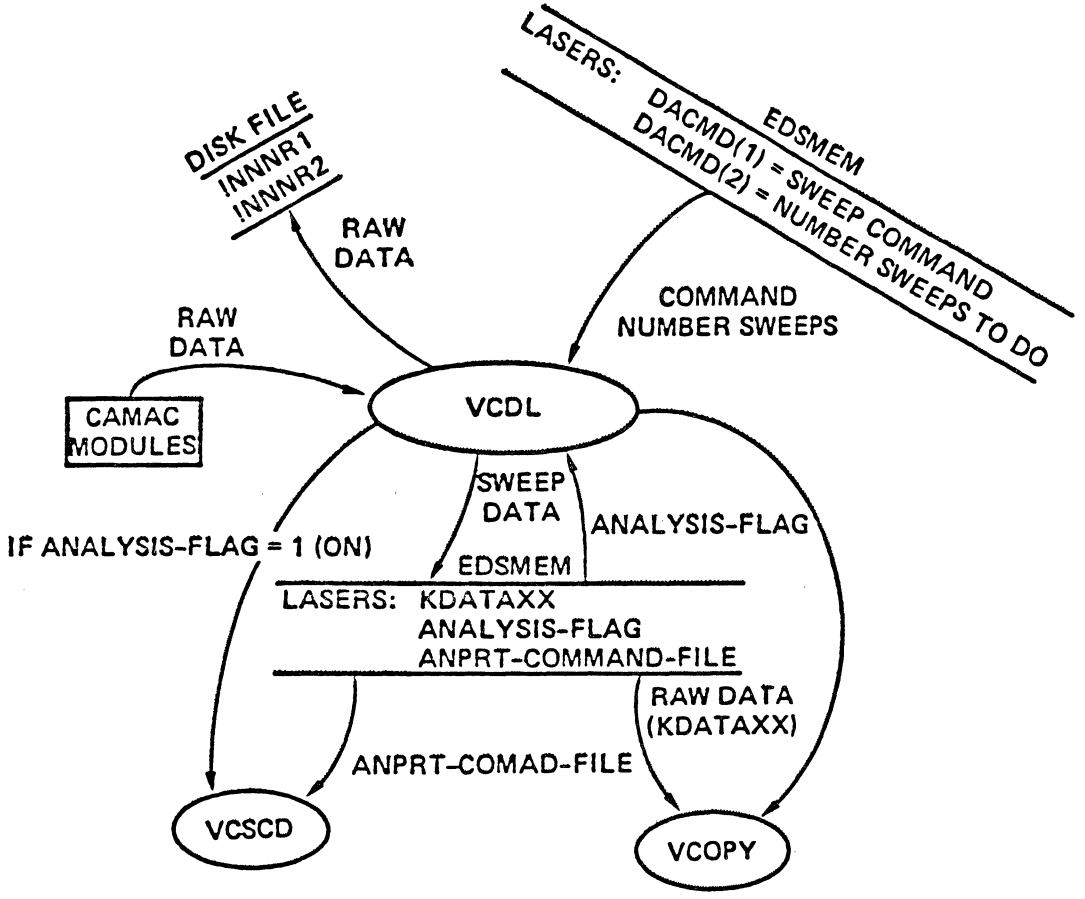


Figure 4. EDS Vapor Sweep Flow.

as rapidly as possible, or at set intervals in a batch mode.

6. The user may adjust a number of parameters during the experiment to tailor the diagnostic to changing needs.
7. End data acquisition.
   The data file is closed, Class I/O terminated.

10. GENERAL SWEEP SEQUENCE

Continuing with the vapor diagnostic example, the sweep command processing is initiated when the user enters a sweep command to the SHELL (Fig. 3). If the user wants to take a number of sweeps as rapidly as possible, SW followed by the number of sweeps to do is entered.

The SW command causes the program VSWEEP to be executed. VSWEEP passes the sweep command and parameters to the data acquisition program, VCDL, via an EMA command stack. VCDL checks EMA for commands to process.

VCDL accepts the sweep command (Fig. 4) and sends the appropriate commands to the CAMAC instruments. When the data is ready to offload from the data logger, VCDL uses the HP routine VMAIO to transfer 32K words of data to EMA. Two VMAIO calls are used to transfer all 64K words of data. If the data is to be archived, it is written to disk and the data base is updated with the sweep information (time of sweep, file name, number of channels, and other parameters).

The graphics sequence is started by scheduling the graphics copy program, VCOPY, with an EXEC 10 call (immediate execution, without wait). In a similar manner, the analysis sequence is started by scheduling the program, VCSCD, with an EXEC 10 call if analysis has been requested.

The "without wait" option permits VCDL to continue without waiting for the son program (VCOPY or VCSCD) to complete. This was one of the requirements, namely, that graphics or analysis would not interfere with data acquisition. By requesting "immediate" execution, the graphics and analysis are skipped if either is still actively processing previous sweep data.

11. GENERAL GRAPHICS SEQUENCE

The graphics sequence is started when VCDL schedules the copy program, VCOPY (Fig. 5). VCOPY checks a flag to make certain that the data from the previous copy has been plotted. If it has not been plotted, VCOPY ends. If the flag is clear, the data is copied into the graphics area for the vapor diagnostic. The scan number is checked before and after the copy to ensure that the data is completely from one sweep. If the data is complete, the plot executive program, PLTEX is scheduled. PLTEX scans the graphics tables to determine which users have requested data from the vapor diagnostic. PLTEX schedules the plotting program, QUICK_LOOK_DRAW, for each user requesting vapor plots. When the data has been plotted, the flag is cleared so that VCOPY is allowed to overwrite the data the next time it is called.

12. GENERAL ANALYSIS SEQUENCE

The analysis sequence is started when VCDL schedules the program, VCSCD (Fig. 6). VCSCD checks a flag to make certain that the data from the previous copy can be overwritten. If it can not be overwritten, VCSCD ends. If the flag is clear, the data is copied by ANCPY into the vapor analysis EMA work area. The scan number is checked before and after the copy to ensure that the data is completely from one sweep. If the data is complete, the program RTGEN is scheduled. RTGEN does the calculations requested and puts the reduced values in the analysis spreadsheet matrix. The data base contains directives of what to compute, how to compute it, and where to store the results in the spreadsheet matrix. These directives are stored in EMA for RTGEN to use. RTGEN clears the overwrite flag to permit new data to be copied. If plots have been requested of the reduced values, ANPRT is scheduled. ANPRT is capable of sending data to other nodes in the system. ANPRT is an interpretive code which receives its input from a file. The user can change the analysis parameters during the experiment by altering the input file to ANPRT. VCSCD schedules ANCPY, RTGEN and ANPRT with wait. In this way, a new analysis cycle cannot be started until the previous one is completed.

13. BERKELEY SOFTWARE SYSTEMS DATABASE MANAGEMENT SOFTWARE

Since EDS is designed to be data base driven, it was essential to have a quick and reliable data base management system. The Berkeley Software Systems (BSS) hash database management software written by a consultant, Richard Lawhorn, is used for all EDS databases. BSS has been used in a number of projects at LLNL. It uses Class I/O to communicate between the application program and the data base manager. A hashing algorithm is used to extract data from a type 1 disk file by the data base manager program. The disk file is built by a structure initialize program (BSHSI) from a text file, the blocklist. A value initialize program (BSHVI) is used to initialize data cells to defined values. A data base of a thousand blocks takes about two minutes to build and initialize.

14. SYSTEM PERFORMANCE

The use of shareable EMA was the first step taken to increase overall system speed. All frequently used parameters are kept in EMA. If data is to be used in detailed calculations, it is often beneficial to move it from EMA to local buffers. The double word addressing used for EMA variables is slower than the addressing scheme used for local variables.

Critical and frequently used codes were locked into memory and ended with the "saving resources" option. Priorities were adjusted so that the data acquisition codes ran at the highest priority. Priorities of other EDS codes were set according to their importance.

The buffer limits on I/O devices were increased to speed data throughput. Disk accesses were kept to a minimum. Class I/O was used extensively for program-to-program communication. Programs posting a Class I/O call can suspend themselves without CPU overhead and are awakened only when there is a message present. FMP file reads and writes were used instead of FORTRAN I/O since the FMP calls are much faster.

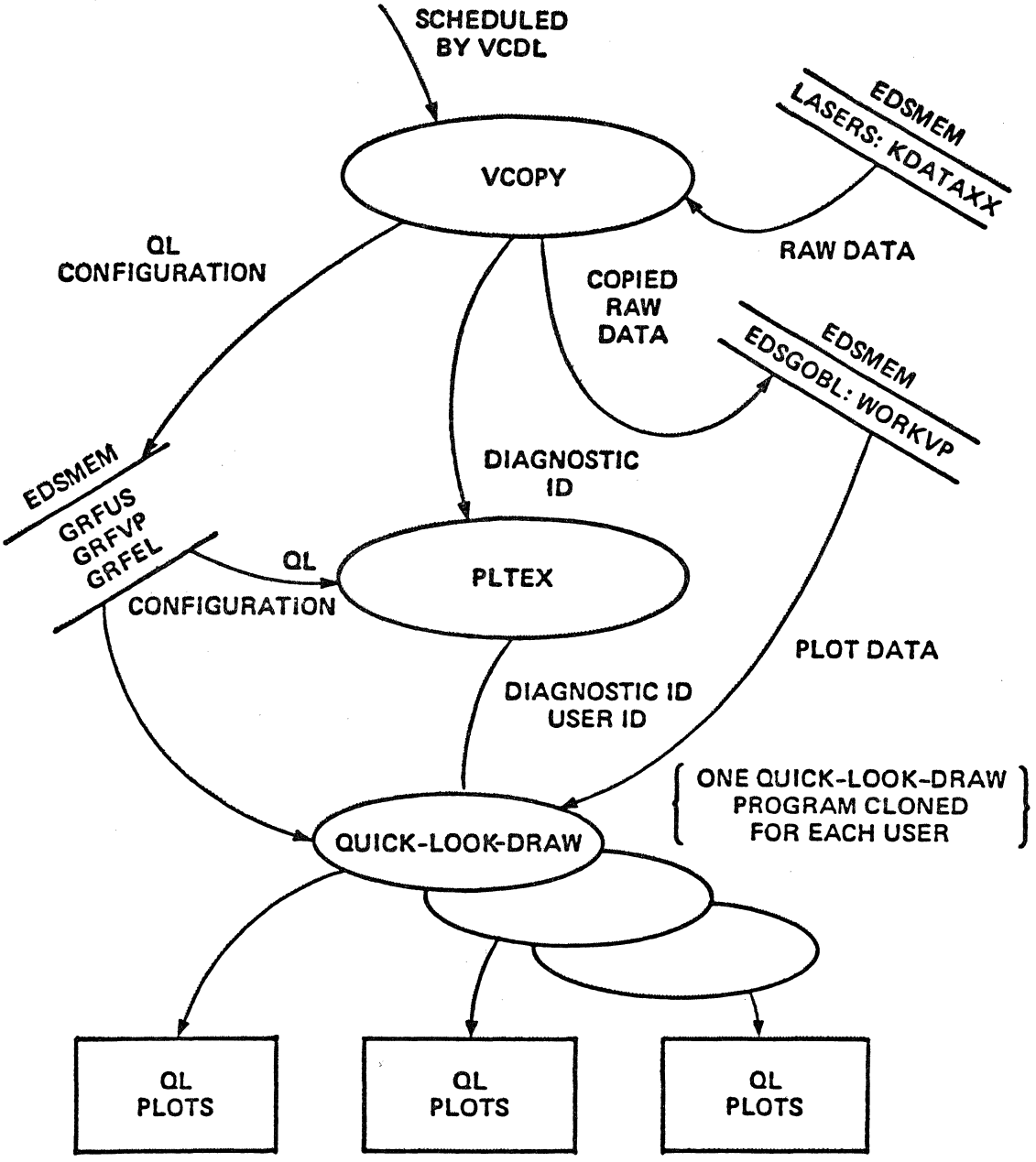


Figure 5. EDS Graphics Flow.

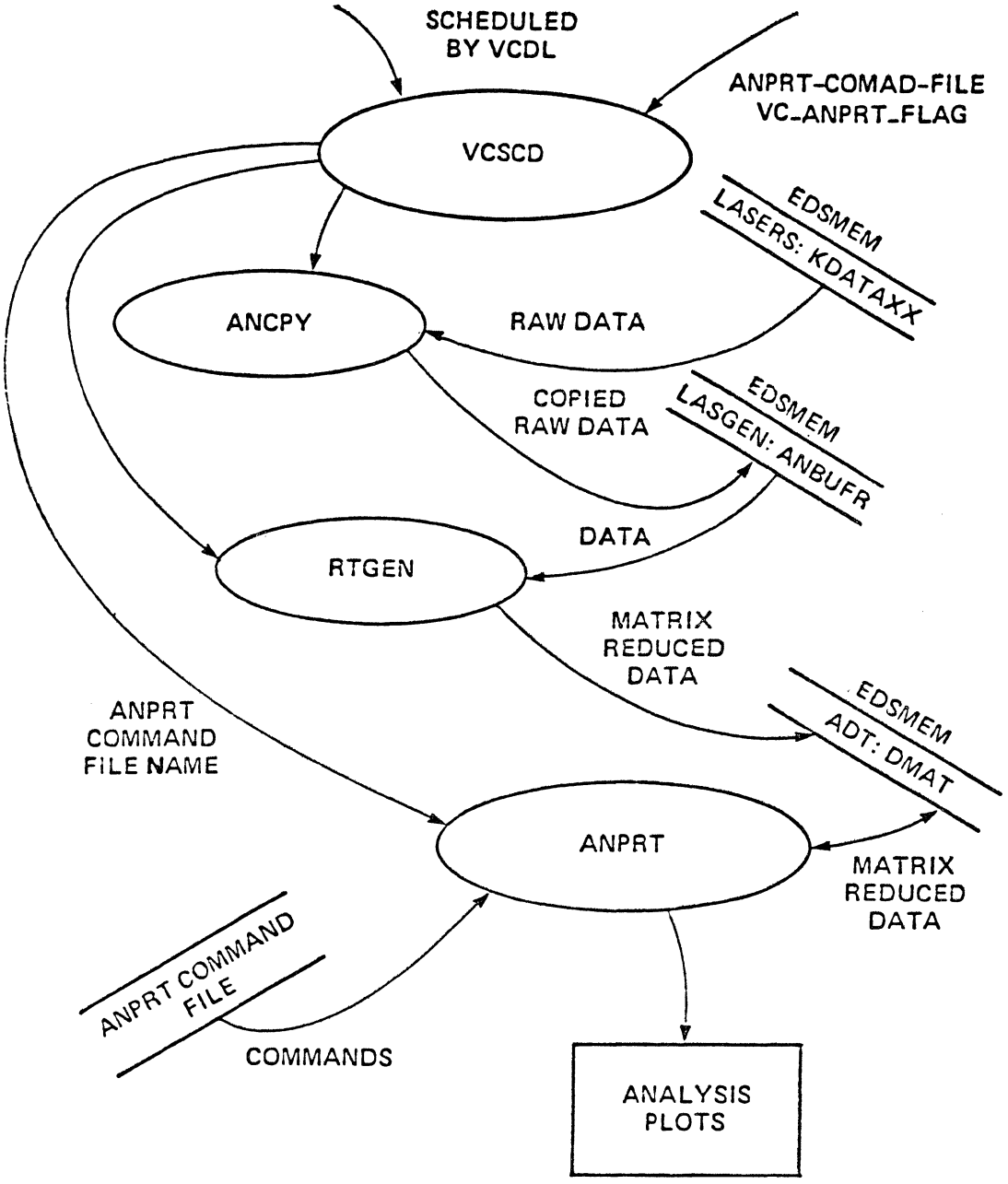


Figure 6. EDS Vapor Analysis Flow.

15. HP EXPERIENCE, GOOD AND BAD

Overall, the use of HP hardware and software has been a good solution. The system has met the major requirements. It was discovered early in the project that the phone-in-consulting service (PICS) was not adequate in resolving many of the problems in a timely manner. A contract was made with HP to provide an on-site consultant one day a week to resolve problems, answer question, perform system generations and write performance tools. This proved to be a very useful service in terms of resolving specific problems. Turning over system generations to HP was less successful in that problems would arise several days after the new system was installed. It was usually necessary to call HP back in to fix the problems. After several episodes of this, the system was frozen and no further updates were done.

Another troublesome area centered around the use of EMA, critical to EDS. The fact that HP provided a mechanism such as shared EMA to permit many programs to have access to large data arrays was very valuable. The restriction that there could only be one EMA declared per program was a problem. Separate common blocks were desired since each program needed only a subset of the total 778 pages of EMA. Two diagnostics, vapor and mass spectrometer, use VMAIO calls to transfer 32K words of data from the I/O device to their respective EMA raw data buffers. The success of a 32K word transfer depended upon the name of the common block into which the data was being written. To add to the confusion, the errors for unsuccessful transfers were undefined.

After many months of investigation by the on-site consultant, it was discovered that LINK was the program that arranged the common blocks in memory and that the user had no control over the arrangement. LINK uses a hashing algorithm on the block names to define memory. The other crucial information obtained was that for a VMAIO call to successfully transfer 32K words of data, the EMA array must begin on a page boundary. In order to work under these restrictions, each common block is padded with the appropriate number of spares to make the block an even multiple of a page. Careful system management is necessary, since the redefinition of any common block requires the recompiling and linking of all codes using shared EMA.

Since the EDS design specifies large data array transfers, a fast copy routine was needed. The VIS EMA move routines seemed to be likely candidates, but unfortunately were not suitable. The VIS routines only work on floating point values, whereas the EDS values are integer. It would be very useful if HP would provide integer VIS EMA transfer routines.

EDS depends upon having only one copy of certain programs active at one time in the systems. Past operating systems (RTE-6/VM and RTE-4B) have included a "don't copy" bit in the program's ID segment to manage cloning. RTE-A has dropped this bit making it much more difficult to control cloning. It was discovered that by linking a code as a system utility and by RP'ing the code with the "don't clone" option, cloning could be inhibited.

Two performance tools provided by the HP consultant were very helpful in finding system bottlenecks. The program CPU displays a CPU utilization profile, and ACT is a detailed system activity monitor. The consultant also provided utilities to programmatically obtain the state of any program in the system.

16. CONCLUSION

EDS has been a successful integration of work done by many people over several years. It is unique at LLNL in the amount of data acquired, the graphics displayed and the analysis done in real-time. The EDS design has withstood the test of time. While continually incorporating new commands, diagnostics, and capabilities, it has been run at regular intervals, once to three times a month. At the present time, EDS has been used in twenty experiments with a total running time in excess of one thousand hours.

17. ACKNOWLEDGEMENTS

I wish to express my appreciation to my co-workers: Steve Telford, Dan Schneberk, Mathilde Killian, Tom Treadway, and George Miller. I thank Jay Ackerman, Joe Brandt, Jim Held and Lloyd Hackel for their support and for the opportunity to participate in the INTEREX 1985 North American conference.

* This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Lab under contract No. W-7405-Eng-48.

1011.

Real Time Analysis under EDS*

Dan Schneberk
AVLIS Program, Y-Division
Lawrence Livermore National Laboratory
U. S. A.

Abstract

This paper describes the analysis component of the Enrichment Diagnostic System (EDS) developed for the Atomic Vapor Laser Isotope Separation Program (AVLIS) at Lawrence Livermore National Laboratory (LLNL). Four different types of analysis are performed on data acquired through EDS, i) Absorption spectroscopy on laser-generated spectral lines, ii) mass spectrometer analysis, iii) general purpose waveform analysis, iv) separation performance calculations. The information produced from this data includes: measures of particle density and velocity, partial pressures of residual gases, and overall measures of isotope enrichment. The analysis component supports a variety of real-time modeling tasks, a means for broadcasting data to other nodes, and a great degree of flexibility for tailoring computations to the exact needs of the process.

A particular data base structure and program flow is common to all types of analysis. Key elements of the analysis component are: i) a fast access data base which can configure all types of analysis, ii) a selected set of analysis routines, iii) a general purpose data manipulation and graphics package for the results of real time analysis. Each of these components will be described below with an emphasis upon how each contributes to overall system capability.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

The analysis software completes the EDS operations cycle which begins with data acquisition and concludes with broadcasting data to PMC-1000 and the Process Modeling node. Two goals define the majority of the work for the analysis component; 1) compute and display meaningful measures of process performance, 2) make all EDS analyzed data available for further analysis or control. To accomplish these objectives analysis software divides into two categories; i) data reduction programs which condense acquired vectors into sets of meaningful scalars, and ii) the routines which make up a spreadsheet-type code (ANPRT) which manipulates, plots, and/or sends data to other nodes in the network.

Data reduction programs are specific to the type of data being reduced. Each program has its own applications work area, and its necessarily different functionality. Four different programs are included in EDS at this time: i) a code for performing absorption spectroscopy on spectral lines from diagnostic lasers, ii) codes which perform gas composition calculations on spectra from residual gas analyzers, iii) general purpose waveform analysis on vectors from the extractor diagnostic, iv) calculations of U235 collected from a combination of different types of data. The results of the different programs are: particle densities, velocities, mole fractions, fraction ionized, evaporation rate, partial pressures of residual gases, stripping efficiency, swu rate.

The spreadsheet-type code (ANPRT), manages the reduced data from every diagnostic. Each applications program has its own columns for writing into the matrix, and its own time vector recording when data was taken and analyzed. Any further plotting or manipulation of the scalar data is performed by ANPRT which can work on separate columns or the matrix as a whole. ANPRT includes: a command interpreter which supports batch and interactive processing, macros and subroutines; a set of intrinsic functions tailored to the needs of the process, a full X-Y plot capability using LLNL Device Independent Graphics Library (DIGLIB), numerous data base functions, and a single command for broadcasting data to other nodes.

To perform this work the elements of the analysis component access the shared data structures in the EDS system and follow the general guidelines for program flow. Like other elements of EDS all analysis software depends upon diagnostic data bases for direction and data identification. The relationship between shared data structures and data base provides the groundwork for many features of the analysis system.

Data Base & Program Flow

EDS data bases do not contain the vector data. Every effort is made to avoid manipulating the data until absolutely necessary. Data Bases do contain the information required for identifying, analyzing and the tracking the data flow through the system. Data bases record a history of data acquisition events. They connect program sequences, vectors and scalars to particular instances in the process.

The data base for the diagnostic lasers is the best example. One particular relation[1] in this data base is exclusively devoted to laser data. Each instance of this relation identifies a different wavelength(frequency), and position for a laser shot (wavelength-position). Of the total items of this type in the data base, only a few are active at any one time, but many different wavelength-positions may be used in the course of process operation. Each item is comprised of four types of information; i) basic identifiers, all the labeling information and parameters for analyzing the data, ii) channel information, the connection between the hardware channels and the software locations of the vectors, iii) time information, the times at which that wavelength-position was used and possibly re-used, iv) analysis pointers, the pointers to the functions to be applied to those vectors and the locations for putting the scalars into the reduced parameters matrix.

Every program sequence depends upon information from the data base at each step, through local tables. Figure 1 presents a pictorial of the five-step EDS sequence; i) acquire vector data, ii) copy to applications workspace, iii) reduce vectors to sets of scalars and put into reduced parameters matrix, iv) manipulate and plot from ANPRT, v) broadcast to other nodes. Prior to sequence execution local tables are filled with the most recent settings for key parameters and pointers. To maintain system speed, no program makes data base accesses during a program sequence. However, all key parameters are in the data base with time stamps attached.

---

[1] This paper subscribes to the definitions and conceptual framework found in E.F. Codd's, ' A Relational Model of Data for Large Shared Data Banks', ACM Communications Vol. 13, (6), June 1970.

Figure 2 contains the pictorial for the laser diagnostics sequence. Labels and parameters for the active wavelength-position items go directly into the applications work area, identifying the vectors of data to the applications program. This includes all the information required to perform the analysis: crossection, wavelength, laser path length, and information for calculating scan lengths. The analysis pointers write a set of flags for directing what function(s) to apply to the vectors acquired, and direct the writing of names and labels to the matrix managed by ANPRT. Re-configuring lasers will result in new values written to local tables, with the episode of use for those parameters and labels recorded in the data base.

EDS Data Reduction Software

While the quantity of vector data handled in EDS is substantial the amount of data reduction code is not large. Fundamentally there are two types of vectors; line spectra, and waveforms. Similar data-analytic problems are common across different diagnostics. There is a base set of component tools which covers the variety of different analyses. With different instrumentation, it is entirely possible that the base set could be made much smaller.

Mass Spectrometer Analysis

Partial Pressures of the different gases in the tank are obtained from this diagnostic. A UTI Residual Gas Analyzer is combined with Lecroy front-end hardware to probe the presence of gases in the low mass ranges. Each sweep of the mass range (1-50 AMU) generates a set of line spectra, the peaks corresponding to the different mass units found in the tank. Figure 3 contains a representative trace of the spectra obtained.

Analysis involves three steps, i) resolving the baseline for the spectra, ii) finding the peak heights of the different mass units, iii) un-cracking the peak height intensities into partial pressures of gases.

Resolving the spectra to a baseline of zero guarantees that peak heights are handled correctly, and allows for easy handling of signal to noise problems. In this task, the first job is to find the baseline. There are two alternatives which combine both speed and accuracy; i) find an unoccupied mass range and compute a mean value, ii) find that value in the amplitude of the trace which is most common by discretizing the range of the trace. The second is more general purpose, and yielded good results for our application. Once the baseline is determined, normalize to zero by subtracting the baseline value from the trace.

There are a variety of techniques for peak detection. Our choice of techniques has been driven by the kinds of problems with input ramp and traces. The sweep over the mass range of 1-50 AMU was not always linear. Traces were sometimes noisy, and mass peaks overlapped enough to make peak heights difficult to determine. Our approach made the most of what we knew. Of the entire range only a few mass numbers were important. Peak detection was restricted to the small ranges of interest. Only peaks above the current signal-to-noise threshold were considered. Overlapping peaks were de-convolved by examining the non-adjacent side of the peak. The limits for the small ranges and the signal-to-noise are in the data base, and could be changed during the run if necessary.

Uncracking the peaks into partial pressures of the gases was performed according to research done by project scientists.

Absorption spectroscopy from Diagnostic Lasers

The data reduction code for this diagnostic computes measures of source performance from absorption broadened spectral lines and related data. A number of different laser wavelengths are mixed into a single shot through the vapor. One absorption broadened spectral trace and a variety of different types of data are obtained for each wavelength. Two different types of variables are computed for a diagnostic laser shot; (i) for each wavelength, state-specific particle densities, widths, (Full width at Half Maximum-FWHM), centers, and amplitudes, (ii) from a combination of these numbers, partition functions, and results derived from in-house models: vaporization rates, mole fractions, fraction ions, and measures of source efficiency.

An absorption broadened spectral line is the result of a particular interaction between laser light and the vapor propagating from the source. Under the conditions obtained in our process, the incidence of a spectral line at that frequency indicates an absorbing quantum transition for certain particles in the vapor.

The mathematical description of absorption broadened spectral lines is a direct application of the differential absorption law[2].

$$(1)\quad I(\nu)/I_0(\nu)) = \exp\,(-\,N_{sl} * L * B(\,\sigma(\nu))$$

or more simply,

$$(2)\quad -\ln(I(\nu)/I_0(\nu)) = N_{sl} * L * B(\,\sigma(\nu))$$

where:

$I(\nu)$ - the laser light exposed to the absorbing substance at frequency ν.
$I_0(\nu)$ - the laser light not exposed to the vapor at the same frequency ν.
N_{sl} - the number particles of the vapor in the lower energy state subject to the transition corresponding to the laser wavelength.
sl - the energy level of the lower state, in wave numbers.
$\sigma(\nu)$ - the absorption crossection at frequency ν.
B() - denotes a functional description of a broadening mechanism for that spectral line.

The conditions in our process ensure the predominance of velocity, or Doppler, broadening. Specific measures are taken to keep other broadening effects at a minimum. The shape of the spectral line reflects the intensity of the flow vectors from the source. The directional velocity of the flow, relative to the laser light, determines the shift in off-center resonance. The velocity distribution of the vapor is written into the broadened line shape. In our case the distribution is roughly Maxwellian, permitting the B() in equations (2) & (3) to be replaced by a gaussian as follows.

---

[2] Mitchell & Zemansky, 'Resonance Radiation & Excited Atoms' Cambridge University Press, Cambridge, England, 1934, contains a full analysis of absorption broadened spectral lines.

$$(3)\quad -\ln(I/I_0) = N_{sl} * L * \left(\sigma(T',\lambda) * \gamma(T',\lambda) / \gamma(T,\lambda) \right) * \exp\left(-4*\ln(2) * \left((\nu - \nu_c) / \gamma(T,\lambda) \right)\right)$$

where:

$\sigma(T,\lambda)$ - is the crossection at center frequency, wavelength λ, and temperature T.

$\gamma(T,\lambda)$ - is the FWHM (full width at half maximum) at temperature T, wavelength λ.

ν_c - the relative center frequency for the line.

T' - the norming temperature for the crossections[3].

The resultant equation is an un-normed gaussian with a number of linear multipliers. Of the many parameters on the right all but N_{sl}, ν_c, and $\gamma(T,\lambda)$ are known. It is the object of the data reduction program to estimate these parameters.

Two different types of line spectra are involved in the analysis of diagnostic laser data; i) a frequency calibration trace from the Fabry-Perot cavity, ii) the absorption broadened spectral line. Both the absorption trace and the Fabry-Perot signal are swept with the same ramp function. While the broadened spectral line contains the particle density and velocity information, it is the Fabry-Perot data which provides the frequency scale for the absorption trace.

Each peak in the Fabry-Perot trace demarcates the position around the center frequency of the absorption line for that point in the sweep, see Figure 3(b). The separation between the peaks measures the distance traveled in frequency for the absorption scan, (ie, 300 mhz, 2ghz etc..). Analysis reduces to; i) resolve a baseline, ii) find the peaks, iii) apply the calibration factor to the distance between the peaks and make the frequency scale.

The best procedure involves Fast Fourier Transforms, but reliable results can be achieved with other techniques. In our work the mass spectrometer routines described above work well for both baseline finding and peak detection. In this case the ranges for finding peaks are determined by the intended scan length dialed into the instrumentation for sweeping the laser. The expected scan length and the calibration factor, are both in the data base and in local tables.

---

[3] Our project scientists chose 2015K as the temperature for reporting crossections.

The frequency scale determined, analyzing the absorption line involves three steps, i) resolve a baseline, ii) transform the data and estimate center, amplitude, and width, iii) compute particle density and mach numbers. Again, techniques described above can work here for baseline fitting. The third task is fundamentally a matter of arithmetic. The majority of the work is involved in computing center, amplitude and width.

While there are many alternatives for estimating parameters in gaussian-shaped data, our work has focused on two techniques; i) order statistic estimators, ii) least squares. The first involves estimating quartiles by numerically integrating data which has the form indicated in (3). The second involves an additional log transform of (3), then a polynomial fit of the result over the peak absorption region. The first technique has the advantage of speed. However, the second is more resilent to common data-analytic problems.

At this time our procedure involves a combination of the two approaches. While obtaining the quartiles involves more work, the total area under the curve described by (3) can be computed in the course of least squares procedures. The estimate of particle density computed with the area total is regularly compared with the estimate from least squares and has proven a better goodness-of-fit statistic than the traditional choices. Estimates of density and mach number take just less than a second per line on an A-700.

More meaningful measures of source performance; neutral densities, evaporation rate, mole fractions etc, are computed from sets of single-state densities. Provided two different transitions for the same substance[4] can be mixed into the same shot, standard equations[4] can be applied to compute neutral densities for any substance in the E-Beam source. Combined with a model of vapor propagation, neutral densities can generate all the measures necessary for mass balance and real time film flow modeling.

The extra computations for these results is not large. Neutral densities are obtained from partition functions (a table of statistical uncertainties, and energy levels for the different energy states for a substance). Also, it has been our experience that modeling results can be approximated by polynomials, minimizing the need for lengthy numerical routines.

---

[4] We make the assumption of internal thermal equilibrium and employ the equations for relating energy-state populations which conform to a Boltzmann distribution.

In our system partition function routines exist in two places, in the data reduction programs, and in ANPRT. The applications program can perform the operations faster but without the same flexibility. ANPRT, running from a command file, has access to a variety of modeling results, and can estimate certain parts of equations if trouble develops with a certain laser. This capability has proven invaluable for a variety of untoward conditions, and has made real-time modeling an oft-used feature.

Extractor Waveform Analysis & Separator Performance Calculations

Calculations for both of these diagnostics is performed within a small interpreter with intrinsic functions for combining columns of the applications work area. Key scalars are written to the ANPRT matrix, but the amount of data reduction for this diagnostic is small, and still being developed.

At this point in time waveform analysis is oriented towards computing received current loads on certain parts of the extractor hardware. This can involve some algebra between waveforms taken at different times, but will always involve some quadrature integration of the ion current waveform itself.

Measures of U235 separation performance integrate data from a variety of different diagnostics. Recieved current loads are taken from the extractor diagnostic. Vaporization rates and source efficiencies are provided by the diagnostic lasers. Once this data is moved to the correct applications work area, the calculations are straightforward combinations of columns of data.

ANPRT Capability

The spreadsheet-type interpreter 'ANPRT' has five components, i) command interpreter with macros & argument passing, ii) a processor for equations and if-conditionals equipped with a set of intrinsic functions tailored to real-time modeling needs, iii) full X-Y plot capability to any viewport or device in the system, (all graphics plotting is performed with DIGLIB), iv) a variety of data base functions for manipulating the matrix.

The reduced parameters matrix contains the analyzed data from the data reduction programs and the names and labels from the data base. The central data structure is a real valued matrix, each row a different instance of a data reduction code, the columns a different computed quantity. In addition to the

separate time vectors written by each data reduction code (decimal time relative to start of the run), there is a six-integer time array, one for each case, providing an unambigous time reference for the data. There is a 4-character ascii identifier for each case for further ease in addressing case ranges. Each real-valued column has an associated 16-character name and 40 character label. Figure 4 is a pictorial of this structure. At present the real valued matrix is 500 by 150. Once 500 events have occured the data reduction programs 'wrap-around' the matrix starting at 1.

The reduced parameters matrix resides in shared EMA and can be accessed by many versions of ANPRT. Each applications program keeps track of the number of cases it has written into the matrix in separate EMA variables locked into memory. Each version of ANPRT has local pointers for that session which can access the EMA pointers for each diagnostic on command. In this way there can be many versions of ANPRT which can operate on the same matrix independently.

The command interpreter combines two types of interpretation in the same general structure. Commands begin with a major keyword which identifies the type of interpretation applied to the rest of the command. The major keywords 'CALC' or 'IF' can be followed by equations or if-conditional statements conforming to the rules in BASIC. All other key words are followed by sets of keywords and parameters. Continuation lines and comments are supported. Macros and subroutines are accessed with the 'TR' command for compatiblity with HP-1000 'CI' and 'FMGR' usage. Menus arise as prompts when not enough parameters are supplied for the keyword oriented syntax. In this way the range of different users are accomodated. Experienced users can suppress menus once commands are known, while novices have the benefit of on-line tutotrial help in the menus.

The processor for equations and if-conditionals features a number of built-in functions. This type of interpretation parses expressions into reverse-polish strings of function addresses and data addresses. The two subroutines which perform the work are driven by the tables of variable, scalar, and function names. As mentioned above the syntax for operators and arguments conforms to BASIC-1000 in all major respects. Arguments are columns in the matrix; or user-defined scalars.

Any column in the matrix can be addressed by 16-character name or by a 'VNNN' identifier where 'NNN' is the column number in the matrix, (name or 'SNN' for scalars). The intrinsic functions are selected specifically for the needs of the experiment. In addition to the standard set of functions[5], a complete set of routines is supplied to compute energy state populations for any component in the vapor. Intrinsic functions exist for computing internal temperature, total population, or any specific state population. Figure 5 contains a list of some of the intrinsic functions.

All graphics are implemented with DIGLIB and drivers exist for HP 26XX terminals and Raster technologies graphics devices. While the plotting is performed with one command, 'PLOT', the variety of graphics settings are manipulated with the 'SET' major keyword. To plot any variable(s) the 'PLOT' major keyword is issued followed by one or two variable names or 'VNNN' identifiers, i.e. 'PLOT V5 URATE'. If only one variable is supplied the X-axis is the index of the variable. Every plot is performed with the extant graphics settings for: axis scaling, line type, point style, device, grid display type, color, superpose options, titles, and hardcopy options. Until a new setting is supplied the current value remains in force. Figure 6 has a list of the graphics commands.

Data base operations are performed primarily through other keywords, with case pointers and variable pointers as parameters to those keywords. With these commands it is possible to delete columns, rows, restrict the case range for 'CALC' operation, enter data directly into the matrix, etc,.. With these commands it is possible to focus on any column or row in the matrix.

Curve fitting is also included in the package; polynomial fitting and general purpose linear regression. These commands can prove useful in real-time or in post-run analysis.

From the command interpreter, one row of data can be sent across DS-1000 to PMC with the 'SEND' command, i.e.,(SEND VARS V3, URATE, V4-V7 will send variables 3,4 to 7 and the variable 'urate' for the specified case). Once this command is issued a master program is initiated which ships real values to the slave on another machine. In the event of any problem both the master and slave terminate without affecting ANPRT operation. The user-specified names of the columns are the parameters for the 'SEND' command. Figure 7 contains an example of commands used during process operation.

---

[5] By 'standard functions' I mean the set found in the variety of different languages, SIN,COS,ABS,ACOS,ASIN etc..,.

Conclusion

The EDS Analysis system contains two types of software; data reduction programs and a spreadsheet-type code. They combine to form a system which generates reliable estimates of source performance, extractor performance, gas composition and overall process performance. Further, EDS analysis can support a variety of real time modeling tasks, and bridge the gap between EDS and PMC-1000. EDS is a general solution to the problems which will arise for real time systems that combine scalar value periodic data acquisition with high-speed vector oriented data acquisition and analysis.

EDS analysis is both flexible and powerful. At any one time the analysis of the vector data can be directed or re-directed from the data base. Different types of data reduction can be performed by changing settings in the data base. Once within ANPRT, new variables or more sophisticated measures of system operation can be computed interactively or through a command file. Indeed all the kinds of modeling performed in ANPRT can be performed on an automatic basis. More than one ANPRT session can operate at one time supporting a variety of analysis and monitoring needs during the experiment. The advantage of full data base capability and filtering controls the data entry into PMC on the basis of values computed from other moving quantitites, assuring the integrity data input to control algorithms. The variety of X-Y graphics to any viewport presents a picture of system response over the history of the process.

This effort takes exception to other approaches to real time systems that uniformly sacrifice flexibility for supposed 'speed of operation arguments'. EDS shows that a great deal of flexibility and power can be obtained in a real time system oriented to vector data, within the time intervals assumed by many commercial process monitoring and control systems.

In spite of the fact that EDS can work without operator intervention in fully automatic mode, it is not a desirable plant system. There are disadvantages to too much flexibility. Eventually EDS will be replaced by a number of devices attached to a much larger control system. EDS is an intermediary system which can take a process from experimentation to plant operation. The success of the AVLIS program underscores this point.

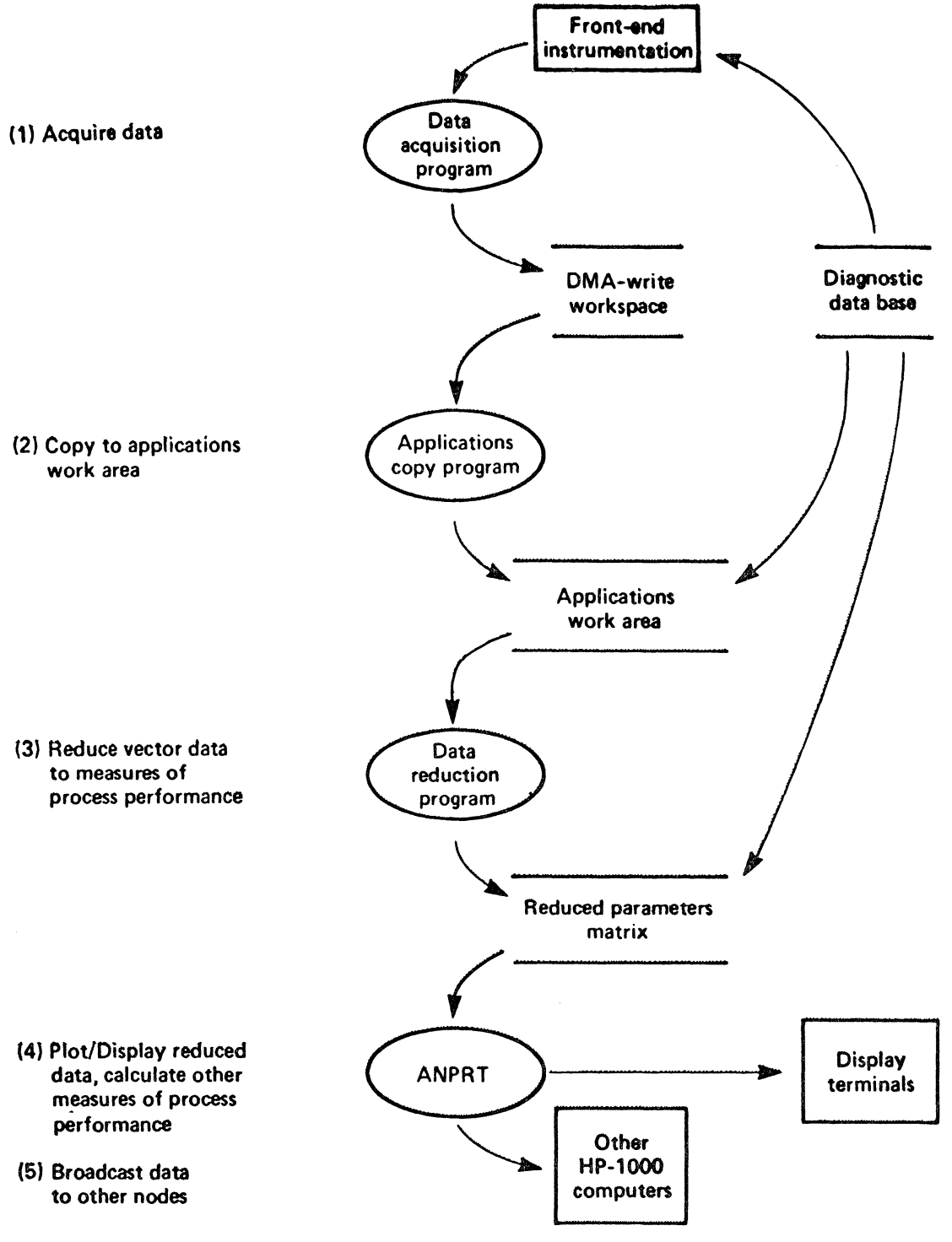


Figure 1. Generalized EDS sequence

(1) Acquire raw diagnostic laser data

(2) Copy data to lasers workspace

(3) With data base parameters, reduce data to densities & widths

(4) Calculate accumulated totals

(5) Broadcast uranium data to mass balance code and PMC-1000

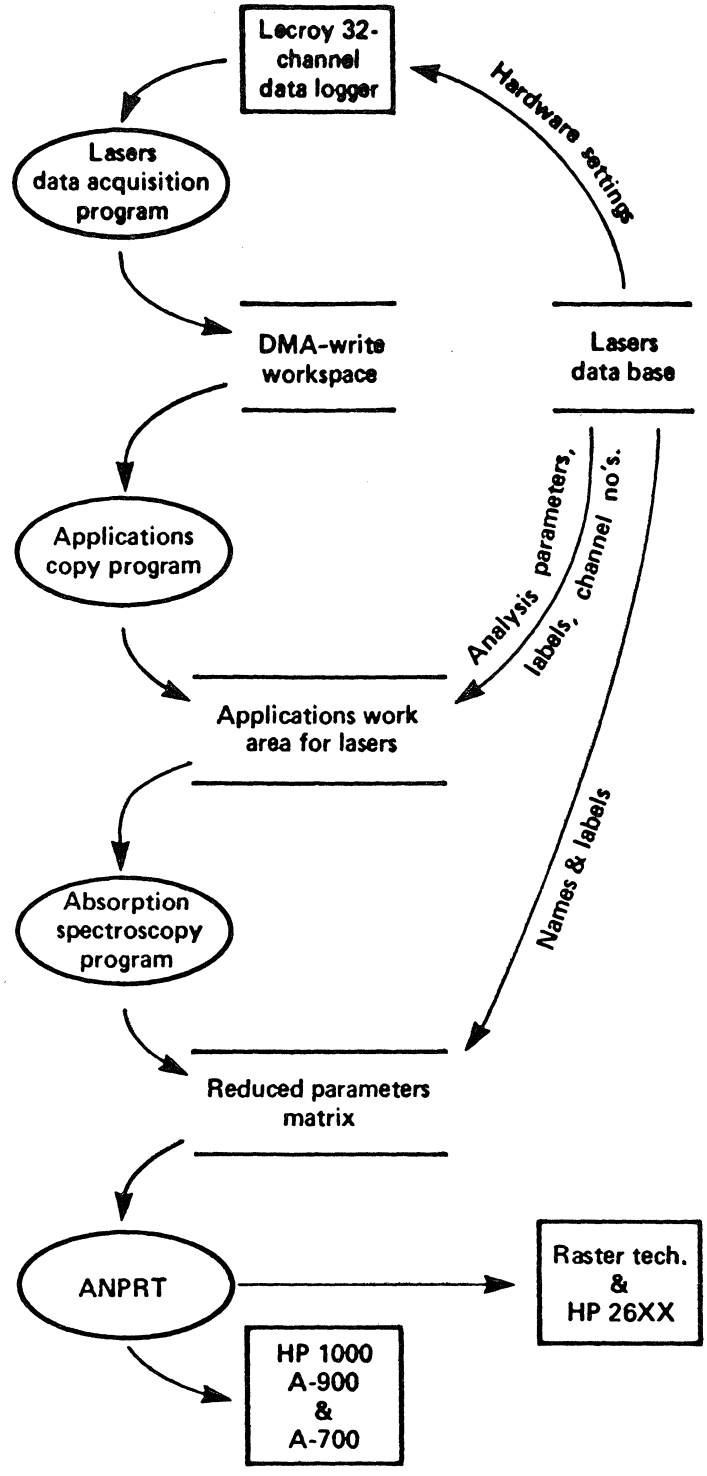


Figure 2. EDS sequence -lasers

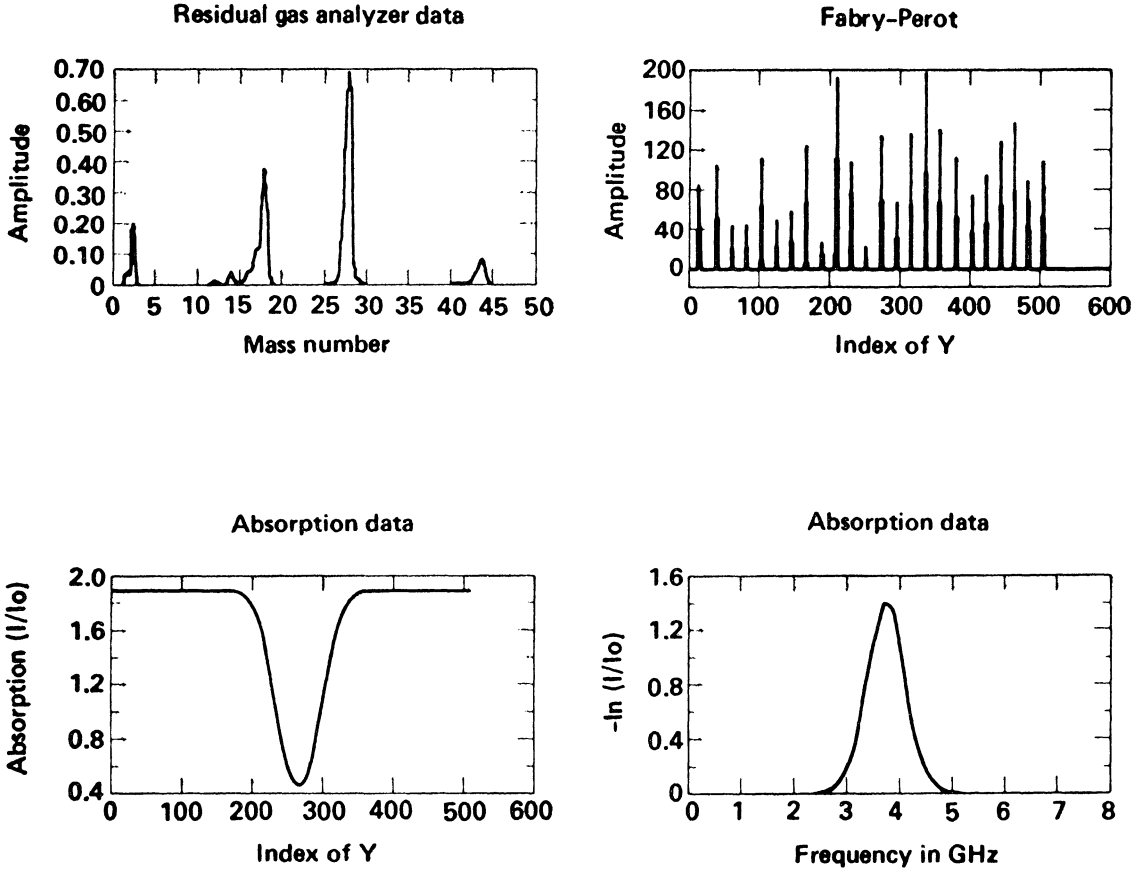


Figure 3. EDS vector data

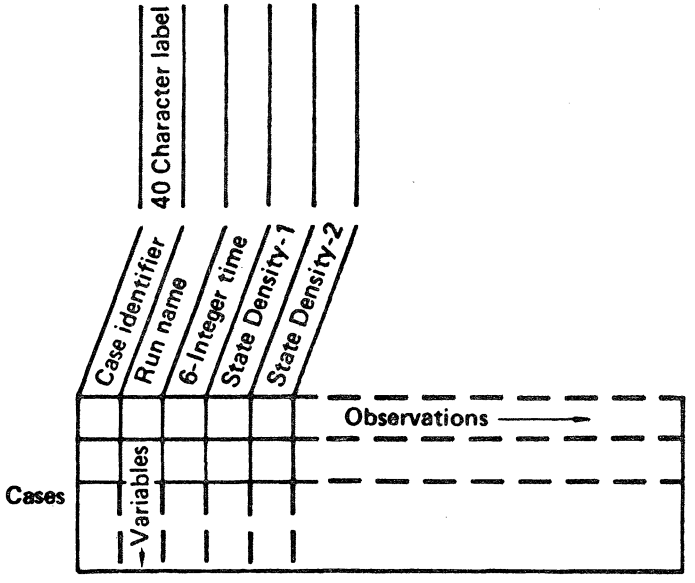


Figure 4. Reduced parameters matrix

ANPRT INTRINSIC FUNCTIONS

Function	Description	Example*
LOG	Computes Natural Log	V3 = LOG(V4 + 9)
EXP	Exponential Function	NEW1 = EXP(V23)
SIN	Sine Function (in radians)	V10 = SIN(V6)
COS	Cosine Function	V11 = COS(V21 * 0.44)
TAN	Tangent Function	V16 = TAN(theta)
SQRT	Square Root	SCAL1 = SQRT(4.56)
SUM	Calculates sum of V10 over specified case range	SCAL2 = SUM(V10)
ABS	Absolute value	V3 = ABS(V14)
ASIN	Arc-Sine function	V10 = ASIN(V13)
ACOS	Arc-Cosine function	V13 = ACOS(V104)
ATAN	Arc-tangent function	V12 = ATAN(V14)
INDX	Puts case index into V8	V8 = INDX
GRAB	Grabs variable V3 from case 1 to current range	V12 = GRAB(1,V3)
ERF	Error Function	V15 = ERF(V6)
SMTH	Performs 11-point quadratic smoothing	V16 = SMTH(V15)
TRAP	Trapezoid rule integration on V10 using V3	V15 = TRAP(V3,V10)
RIEM	Riemann sum on V11 using V3	V19 = RIEM(V3,V11)
TENEUT	Computes uranium internal temperature from two state specific densities	V10 = TENEUT(V3,V4,0,620)
NTOT	Computes Neutral Uranium density from one state specific density and a temperature	V12 = NTOT(V3,0,V10)
EXPOP	Computes an expected population from neutral density,temperature and an energy level	V15 = EXPOP(V12,V10,3801)

* - in ANPRT all of these commands would br preceded by the major keyword 'CALC'.

FIGURE 5

ANPRT GRAPHICS SETTINGS

* All graphics setting are accessed with the 'SET' major keyword, combined with the appropriate minor keyword and option

Keyword	Description	Example
VIEW	Select Viewport, there are 8 available	SET VI A
(all settings apply to the selected viewport)		
POINT	Point Style Options, there are 12 styles	SET POINT A
LINE	Line Style Options there are 10 types	SET LI B
AXIS	Axis settings, can set for autoscaling or for explicit axis scale	SET AX Y 0 1 0.1 0.02 (set y-axis explicitly) SET AX A Y (set y-axis to autoscale)
LOG	Log axis option	SET LO A
PEN	Pen color options	SET PEN A
GRID	Turn on grid	SET GRID ON
SU	Superimpose Option	SET SU ON
DUMP	Automatic hardcopy	SET DUMP ON

FIGURE 6

Command File Example

** Command files are sent in the run string with the the call to execute ANPRT. The lines below are a sample for a command file used during a run, for the laser data. The '*' denotes a comment line.

```
*
* Tell ANPRT the diagnostic, REFRESH local tables, and set to
*  operate on last case
IAM VAPOR
REFRESH
SET CASES $,$
*
* Compute Uranium internal temperature from ground and 620
* state-specfic densities
CALC UTEMP = TENEUT(V4,V6,0,620)
*
* Calculate Neutral U-density,from ground and temperature
* and calculate URATE with UFACT function from modeling work
CALC UTOT = NTOT(V4,0,UTEMP)
CALC URATE = UTOT * UFACT
*
* Accumulate URATE from beginning of data
SET CASES 1,$
CALC U_ACCUM = RIEM(TIME,URATE)
*
* Send data to other nodes
SEND VARS UTEMP,UTOT,URATE,UACCUM
*
* Plot last 40 cases on viewport A
*  Auto scale data for X and Y Axis
*  Filter data with IF statement
*  Set Point style,and Line style and plot
SET CASES -40,$
SET VIEW A
SET AXIS AUTO BOTH
IF (URATE GT 0) AND (URATE LT 110)
SET POINT I
SET LINE A
PLOT TIME URATE
ENDIF
**
** all done
EX
```

FIGURE 7

1012. FILE BACKUP AND FILE ARCHIVING ON THE HP1000

George E. Santee, Jr.
Glen A. Mortensen
Intermountain Technologies, Inc.
P.O. Box 1604
Idaho Falls, Idaho 83403-1604

FILE BACKUP AND FILE ARCHIVING

File backup and file archiving both are intended to reduce the possibility of data loss on a computer system. Common reasons for data loss include user errors such as editing an existing file to make a revised copy and then over writing the original file instead of creating a new file, and writing to an existing magnetic tape instead of reading from the tape. Hardware failures such as a disc head crash can also result in loss of data. Another form of data loss occurs when a user either forgets the name of the file that contains the data or forgets what data is in a file. Most computer system managers back up their systems to magnetic tape frequently enough that should some event cause destruction of some files, these files could be restored. However, system backups are generally recycled at regular intervals. Therefore, if a user has removed a file from the system for a period of time greater than the recycle interval, the user needs a different type of backup, a form of backup that has an extended lifetime. This form of backup is commonly called archiving. When information is archived, the user generally intends to maintain the data in an accessible state for an extended period of time. Archived information can be restored to a system without any conflict with other system functions whereas backup information generally replaces the entire contents of a system when restoration occurs. If a computer system manager adopts an archiving philosophy instead of a backup philosophy, the computer users can enjoy the benefits of backup, i.e., being able to recover from losing a large number of files, as well as the benefits of archiving, i.e., being able to recover individual files. The only requirement that must be met for an archiving system to replace a backup system is that the archiving must be done at regular intervals rather than only at times when someone wants to make an archived copy of a file. Thus, if archiving is performed regularly, the system backup becomes a subset of the file archiving system.

DEFINITION OF TERMS

ASCII - The American Standard Code for Information Interchange. The ASCII code uses 7 bits to represent 128 possible characters on a computer. ASCII is used on the HP1000 for all character oriented data storage.

EBCDIC - Extended Binary Coded Decimal Interchange Code. The EBCDIC code uses 8 bits to represent 256 possible characters on a computer.

Inter Record Gap - The space between physical records on a magnetic tape unit. (Approximately 0.6 inches).

Inter File Gap - The space between files on a magnetic tape unit. (Approximately 3 inches).

Blocking - The process of combining a number of logical records into a single physical record.

Logical Record - The smallest collection of data that can be accessed by a user.

Physical Record - A physical record contains one or more logical records. The physical record is the amount of information that is transferred to or from a storage device.

Track - That portion of a disc surface that is covered by the disc head during one revolution of the disc surface.

Sector - The smallest division of a disc track. On the HP1000 a sector is 64 words.

Directory - The set of information which describe a file's attributes. This information includes such items as file name, file size, file protection, and record length as well as the location of the file.

AVAILABLE MEDIA

There are a number of different types of media available for archiving information. Among the options are Magnetic Tape, Magnetic Disc, Optical Disc, Paper, Microfiche, and Punched Cards.

Perhaps the oldest form of archiving is punched tape and punched cards. This method of archiving as a matter of practical use has a very limited storage capacity.

Microfiche of printed data is another form of archive media. Microfiche is quite durable and can conveniently hold large quantities of data. However, recovery of large quantities of information from this media is very time consuming and prone to transcription errors.

Paper is another method for archiving information. Paper suffers from its bulkiness and the print quality deteriorates with time and exposure to sunlight. Recovery of large quantities of information from this media is very time consuming and prone to transcription errors.

The magnetic disc offers yet another form of archive media. The magnetic disc is a very fast, high density storage medium. Information is easily moved to and from magnetic disc. The major disadvantage of the magnetic disc is cost.

Magnetic tape is currently the most popular form of archive media. Magnetic tape is relatively inexpensive, has high storage density, and is quite fast. A disadvantage of magnetic tape is that it deteriorates with time. Also included in the magnetic tape category are the CS80 cartridge tapes and the 264x cartridge tapes. The 264x cartridge tapes have a limited storage capacity.

The optical disc provides an new form of archive media that will prove to be more widely used in the future. Current information suggests that the optical disc will provide an archive media with an almost infinite life.

IDEAL ARCHIVING SYSTEM

For an archive system to effectively function as both an archive system and a backup system, one can think of a number of highly desirable features. At the top of the list, the archive system should be easy to use. If the user finds the system difficult to use, then archiving will not be done on a regular basis and before you know it something will happen and destroy some of the user's unarchived files. So an archive system that is difficult to use is really of no use. The user will more than likely never have the file archived when it is needed.

The archiving system should also efficiently use the space on the archive media. Not only does this reduce the cost of the archive media, but it also reduces storage problems. In order to efficiently use space, the archive system must allow multiple versions of a file to be stored on the media. The archive system must also be able to pack the data as densely as the archive media will allow.

Another feature of an ideal archiving system is a means for recovery from some of the more common occurrences that might affect the media. For example, a user should be able to recover data even if part of the media is over written. From the point of view of over writing data, the current generation of write once optical discs provide an extremely safe archiving media.

Another desirable feature is to have quick access. Quick access would be possible if the media were on line at all times. However, on-line access introduces a new set of problems unless the archive media is not prone to destruction or unless dual-but-distinct archiving methods have been employed.

Another desirable feature of an ideal archiving system is to have an on-line database associated with the archive media so that the user does not have to keep track of what file is on what archive media. However, this database should not replace the directory that is on each volume. It is essential that each volume of the archive media contain a directory of the files on that media. Ideally, this directory should contain enough information about a file so that a user will know what is in the file at some later date without having to look at the contents of the file.

FILE STRUCTURE ON HP1000

The file structure on disc can be important in some archiving systems, e.g., a system that makes a copy of the disc image of the information. For the HP1000, the disc contains a directory either in file manager, FMGR, format or CI format. This directory maintains the file attributes such as size, record length, number of extents, name, security code or protection status, owner, and track and sector disc address pointer to the file. The actual format of the data on disc depends on the file type. For type 1 and 2 files, the disc file record format consists of 128-word blocks. For a type 1 file, the record length is equal to the block length of 128 words. For type 2 files, the record length is user defined and may cross block boundaries.

For files of type 3 and greater, the records are of variable length. The variable length disc file record format consists of a record length word followed by the data which in turn is followed by a redundant record length word. Sub-file

marks are indicated by a zero length record, i.e., two adjacent zeros with no data. An end-of-file <EOF> mark is indicated by a -1 in the first record length word location.

AVAILABLE SOFTWARE

Six HP supported programs: FMGR, SAVER, FC, TF, WRITT, and PSAVE, and three third-party programs: DUPER, SAVEM, and GSL/1000 will be analyzed in terms of meeting the requirements of an ideal archiving system.

FMGR

For FMGR ST, DU, or CO transfers to tape, each record of the original file is sent to or from the tape as one record. The records may be variable length. Each file is terminated by a single EOF mark. The tape utilization is very low for this format and no directory information is maintained on the archive media. Hence this format is unacceptable for an archiving system. The format for a tape created with FMGR is shown below:

```
File 1
<EOF>
File 2
<EOF>
File N
<EOF>
<EOF> - Added by the CN,lu,EO command
```

SAVER

The SAVER utility was the first HP supported program to save individual files as well as maintain a directory of the files. SAVER works with either half-inch magnetic tape or 264x cartridge tape. SAVER blocks files into records of up to 2048 words (128 words on 264x cartridge tape) and is therefore quite efficient in using the archive media. However, SAVER does not allow multiple versions of a file on tape which limits space efficiency. SAVER is quite easy to use and has an on-line help menu. The current version of SAVER is limited in that file names can be no longer than 6 characters and hence only works with FMGR files, although this will change when SAVER II is released. SAVER does lack features for recovering from one of the more common occurrences that can damage a tape, writing over the front of the tape. If the front of a SAVER tape is over written, all information about file name, size, location, etc. are destroyed. The format of a SAVER tape is shown below:

```
Directory
    Header - 80 bytes
    N File Entries - 50 bytes per record
<EOF>
File 1
<EOF>
File 2
<EOF>
File N
<EOF>
<EOF>
```

Tape file records are images of the disc records including the length words. The records are blocked to a maximum of 2048 words (128 words for 264x cartridge tapes). The last record on tape may be less than 2048 words, but will not be less than 128 words.

FC

The FC utility saves individual files and maintains a directory on the front of the tape. FC blocks files into records of up to 4096 words and is therefore quite efficient in using the archive media. FC has both a local file directory at the front of the tape as well as distributed file directories at the beginning of each data file. However, FC does not allow multiple versions of a file on tape which limits space efficiency. FC limits the length of file names to 6 characters. An FC tape file may contain a number of disc files. The format of a FC tape is shown below:

```
Header File
Gap Check File
Comment File
Volume Body
Directory File
Data Files
<EOF>
<EOF>
```

TF

The TF utility is an extension of the UNIX TAR utility. TF is currently the only HP utility that will work with both the FMGR and the CI file systems. The file directories on a TF tape are distributed along the tape. Hence, it is quite inconvenient to get a directory listing of the file names on the tape. TF allows a user to save files on half-inch magnetic tape or on magnetic disc. TF blocks files into records of up to 5120 words (ten 512 word logical blocks) and is therefore quite efficient in using the archive media. Additionally, TF allows multiple versions of a file on tape which further enhances space efficiency. If the create time for a particular file on a TF tape is not known, recovery of files from a TF tape can be quite time consuming since TF must go through the entire tape looking for file matches. TF does allow file names to be longer than 6 characters, i.e., CI file name format. The two greatest hindrances of TF are the fact that directory lists take a long time, and the fact that file recovery requires that the whole tape be read unless the create time is known. The format of a TF tape is shown below:

```
Directory Blocks
Data Blocks - No data blocks follow the directory
              block for a directory file.
End-of-Data Blocks - "TF End of Tape padding block"
Padding Blocks - Fill remainder of record after end
                 of data blocks
<EOF>
```

After the last file on tape, two consecutive data blocks of all zeros are written. Any unused bytes at the end of a data block are undefined.

WRITT

The WRITT utility stores data in the form of disc images i.e. track by track. Hence physical records will be either 6144 words or 8192 words depending on whether the disc has 96 sectors/track or 128 sectors/track. WRITT accesses the cartridge directory and saves only those tracks identified as containing active data. Since WRITT is oriented toward saving tracks, it is very inconvenient to recover single files and hence WRITT is not suitable as an archive system. The format of a WRITT tape is shown below:

```
Header (50 words)
Copy of First Directory Track
.  .  .
Copy of Last Directory Track
First Data Track
.  .  .
Last Track Containing Data
<EOF>
```

PSAVE

The PSAVE utility stores data in the form of disc images i.e. track by track. Physical records will be either 6144 words or two 4100 word records depending on whether the disc has 96 sectors/track or 128 sectors/track. Since PSAVE is oriented toward saving tracks, it is very inconvenient to recover single files and hence PSAVE is not suitable as an archiving system. Unlike WRITT, PSAVE saves all tracks on an LU. The format of a PSAVE tape is shown below:

```
Header - 6 records
         28 words
         10 words
         35 words
         38 words
         34 words
         23 words
Overhead Records
        128 words
         20 words
First Track  (Track 0)
Last  Track  (Track N)
```

For discs having 8192 word tracks, two 4100 word records are written for each track. For discs having 6144 word tracks, each track is stored as a 6148 word record.

DUPER

The DUPER utility is from the LOCUS library. Duper blocks the records of a file into 2568 word physical records on tape and therefore uses the archive media quite efficiently. Additionally, DUPER allows files to be added to tape as long as the file name is unique. The ability to add files to the archive media further enhances the use of space on the media. DUPER maintains a distributed file directory at the beginning of each file on the tape. Therefore, to recover

a file, DUPER must search through the tape. DUPER does not allow file names to be longer than 6 characters. The two greatest hindrances of DUPER are the fact that directory lists take a long time, and the fact that file names are limited both in length and occurrence. The format for a DUPER tape is shown below:

```
File 1 - Contains 8 word file directory in first record
<EOF>
File N - Contains 8 word file directory in first record
<EOF>
```

SAVEM

SAVEM is a utility from CSL/1000 (the CSL/1000 tapes are written in SAVEM format). SAVEM writes physical records of 2048 words on tape and hence uses tape quite efficiently. SAVEM maintains the file attributes in directories at the front of each "save" on a tape. SAVEM also contains a 128 word record at the beginning of each tape file to identify attributes of the disc file. Using the concept of "saves", allows additional files to be added to tape. The restriction for adding additional files is that additional files must be maintained in a new "save". In order to get a complete directory list for a tape all of the directories for the individual saves must be accessed. The concept of saves also makes it necessary for the user to keep track of what version is in each save. An additional feature of SAVEM is its ability to manage the disc file directory in order to speed the rate of file transfer. The format of a SAVEM tape is shown below:

```
SAVE 1
      Directory
      <EOF>
      Directory
      <EOF>
      File 1 in SAVE 1
      <EOF>
      File N in SAVE 1
      <EOF>
      <EOF>
SAVE N
      Directory
      <EOF>
      Directory
      <EOF>
      File 1 in SAVE N
      <EOF>
      File 2 in SAVE N
      <EOF>
      <EOF>
      <EOF>
```

A GSL/1000 tape is extendable, i.e., the tape's table of contents is automatically updated whenever a new file is saved on the tape. An eight-level naming structure, or pathname, allows logical organization of similar files. Up to eight names, including the FMGR file name, can be associated with a file when it is archived on tape. Duplicate pathnames are allowed and are given successively higher version numbers to distinguish them. Any version of a file can be easily restored by specifying its pathname and version number. Multiple copies of the table of contents are stored on the tape, two at the front and one at the end of the data, in order to minimize the chances of losing the information stored there. Even in the unlikely event that all of the copies of the table of contents are destroyed, individual datasets can be retrieved with program RRR. The format of a GSL/1000 tape is shown below:

```
Table of Contents Copy 1
<EOF>
Table of Contents Copy 2
<EOF>
Space for TOC to Expand
<EOI>
First Data Set
<EOF>
Second Data Set
<EOF>
Last Data Set
<EOI>
Table of Contents Copy 3
<EOF>
<EOF>
```

<EOF> = End-of-File mark
<EOI> = End-of-Information mark = <EOF> + (1 record = "EOI") + <EOF>

The table of contents and datasets are written in blocks of 2048 words when possible. Shorter blocks are always a multiple of 128 words. The first 128 words of each dataset make up an ID block which contains enough information to restore the original disc file from the following tape file. The remaining blocks in the dataset are copies of the corresponding blocks in the original disc file.

SUMMARY OF ARCHIVING UTILITIES

	orientation	extendable	directory	block (words)	on line database
FMGR	file	yes	none	variable 2 - 128	no
SAVER	file	no	local	2048	no
FC	file	no	local and distributed	4096	no
TF	file	yes	distributed	5120	no
WRITT	track	yes	N/A	6144 - 96 sec/trk 8192 - 128 sec/trk	no
PSAVE	track	yes	N/A	6144 - 96 sec/trk 4100 - 128 sec/trk	no
DUPER	file	yes	distributed	2568	no
SAVEM	file	yes	local and distributed	2048	no
GSL/1000	file	yes	local and distributed	2048	no

ACKNOWLEDGEMENTS

The authors wish to thank Mr. Bill Hassell and Mr. Dave Doxey of Hewlett Packard for their assistance in supplying information to be used in this paper.

REFERENCES

Hassell, Bill. "Saving and Restoring Data for 1000 Computers." Tutorial for the INTEREX European 1000 Conference. April 9-11, 1985. Antwerp, Belgium.

1013. USING AN HP1000 A-SERIES COMPUTER AS AN SNA GATEWAY

Reid MacGuidwin
Forest Computer Incorporated
1749 Hamilton Rd.
Okemos, MI 48864

INTRODUCTION

Forest Computer specializes in developing and supporting software for very high performance transaction processing and networking on HP computers. Increasingly, these disciplines are required in multi-vendor environments. A large percentage of these multi-vendor sites combine IBM/SNA networks and one or more other vendors such as Hewlett Packard, Burroughs, Prime or DEC.

In November of 1984, Forest approached a large commercial loan corporation confronted with such a multi-vendor communications problem. This case required a communications gateway between 600 existing Burroughs terminals and printers and a newly installed IBM 3083 mainframe and its IBM 3725 front end processor.

This particular site posed some interesting problems. The company had an installed base consisting of some 150 terminals located at the home office in Chicago. The remaining 450 devices were spread among 10 branches in the United States, Canada, and Europe. The client had decided to migrate the existing application systems from Burroughs to IBM, so for some period of time the users would need access to both vendor's mainframes. Faced with the need to provide their user base with this dual access, the company came very close to the extremely costly undertaking of placing two terminals (the existing Burroughs and a new IBM) on each user's desk, and installing separate leased lines to each branch to handle the IBM traffic.

Forest, having already experienced such dilemmas at several other client sites, presented the company with four possible solutions which would allow one terminal to serve both host processors. These four options were: IBM Series I minicomputers, Network System's Hyperchannel product, protocol convertors, and a protocol converter/terminal emulator running on an HP1000 A600+ using Forest's proprietary software.

The costs of the first three options ranged from $125,000 (plus a good deal of IBM systems programming) for the protocol convertors, to $300,000. The fourth option could be made to provide the same services at a fraction of the cost, while requiring no modifications to the IBM system and permitting a "vanilla" SNA environment.

Forest had HP1000-based products which had been used previously in similar circumstances where specialized communications were called for. It was shown that these could be enhanced to meet the customer's needs within the implementation schedule of 10 weeks.

HARDWARE

Forest chose to base its products on the HP1000 A-series for several reasons.

The RTE-A real time operating system is well suited for data communications applications. In addition, the A-Series support of Programmable Serial Interface (PSI) boards makes this computer an ideal candidate for protocol development. In this case it would be required to handle both the Burroughs and IBM SDLC protocols. The capability of RTE-A to support user-written drivers enabled the high performance that we would require to support the message volumes generated by the 600 devices. Finally, EMA (Extended Memory Arrays) support under RTE allowed us to keep all working tables in memory, again to maximize performance.

The hardware configuration consists of an A600+ with 3M bytes of memory, a 7941A 24 Megabyte disk drive, a 9144A cartridge tape drive, and two 12005 asynchronous interface cards (one for the system console, the other for remote, dial-in diagnostics). The final hardware components were the three PSI cards which handle communications between the Burroughs FEP, the A600+, and the IBM 3725.

PSI BOARDS

The PSI board is a programmable data communications interface card. It contains a Zilog Z80 microprocessor with 16K bytes of RAM and supports up to 16K bytes of ROM. It has Z80 peripherals including two DMA chips, a 4 channel Counter Timer Chip (CTC) and a two channel Serial I/O chip (SIO). There are also several memory mapped latches for control of the data communications channels and the backplane interface. The backplane is the standard L/A-Series custom I/O processor (IOP) chip which includes on-board DMA access to the A600+ memory system, plus all backplane handshake signals. The backplane appears to the Z80 as an 8K byte segment of memory. The Z80 DMA can be used with the IOP DMA to transfer from PSI RAM to the A600+ memory without interaction by either the Z80 or the A600+ processor.

DATA COMMUNICATIONS CONSIDERATIONS

Although initially the system would have to handle only moderate message throughput (approximately 1 message per second), we realized that ultimately, as the application load shifted to the IBM, the gateway would have to handle volumes closer to four messages per second. Due to line speed limitations on the Burroughs FEP, the fastest link we could provide between the FEP and the A600+ was 19.2K baud. We chose the Bisync (BSC) 3780 contention protocol for maximum throughput. Since this speed allows only one full screen (1920 character) message per second, we initially configured the system with four of these lines between the A600+ and the Burroughs.

On the other hand, the IBM 3725 supports SDLC line speeds to 56Kbps. Since this permits message volume close to eight full-screen messages per second, we were confident that a single 56K baud link would handle the volume requirements. Figure 1 depicts the datacomm configuration of the various computers.

SOFTWARE

As a rule, we generally program all protocol and line handling functions in Z80 assembly on the PSI card. Forest has created a proprietary operating system on the Z80 specifically designed for datacomm applications. By offloading the CPU intensive chores of character transmission and reception, error handling and retries, and upping and downing of devices to the PSI card, the HP1000 can be

FIGURE 1 – DATACOMM CONFIGURATION

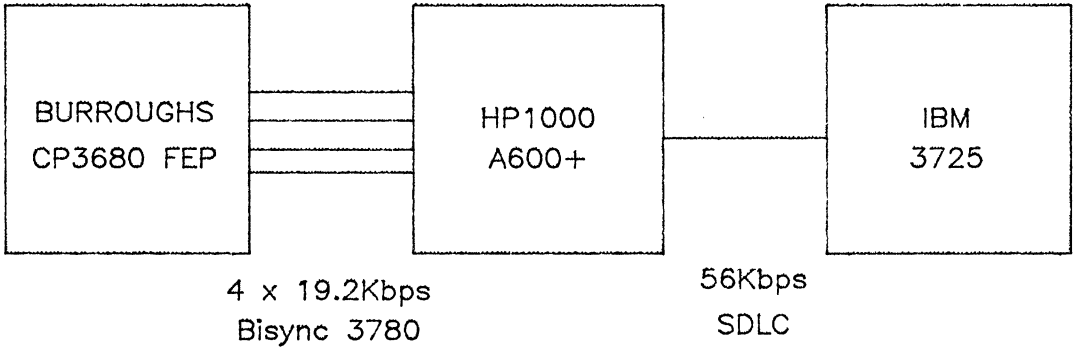
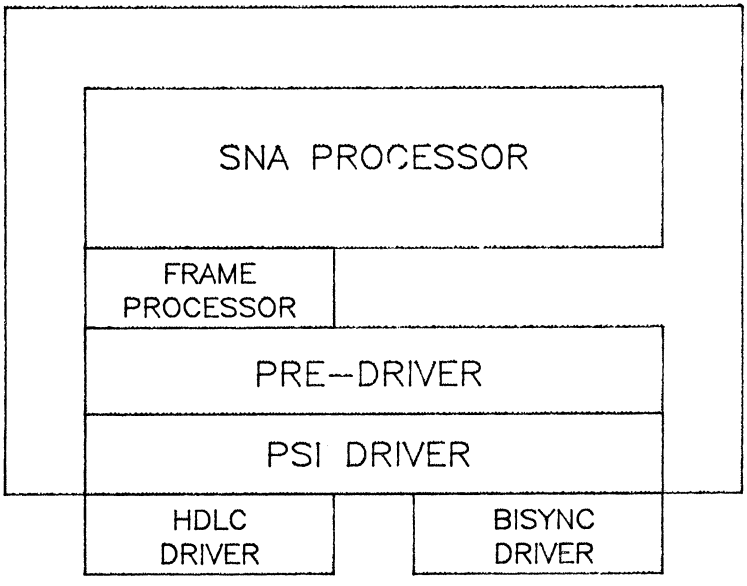


FIGURE 2 – SOFTWARE MODULES



turned into a high-powered message switch, capable of "front-ending" terminal networks in excess of 1000 devices.

For this project we decided to put only the lowest level of the SDLC protocol, transmitting and receiving frames, on the PSI card. Two considerations caused us to initially forego developing the full SDLC protocol on the PSI card. First, we were faced with a ten-week deadline for installation. Second, although we had created the PSI operating system, we had not required line speeds in excess of 19.2Kbps before this project. To err on the side of caution, we therefore chose to limit processing within the PSI card until we were sure that the card could handle the 56Kbps speed required to the IBM.

Our initial design, therefore called for most of the SNA processing to take place within the HP1000. Figure 2 shows the individual tasks that comprise the system. Tasks communicate with each other through the shared memory facilities of RTE-A. We have written a set of procedures to simplify the efforts of this form of intertask communication.

Following, is a brief description of each task on the system.

HDLC DRIVER

Bit-oriented support on the Z80 SIO chip made this task relatively easy. By properly initializing this chip, the PSI card will handle flag recognition, character framing, zero insertion and checksumming. Thus, within our existing PSI operating system, we were able to quickly develop a streamlined HDLC line driver which passed all inbound messages to the A600+. A small header on each message told the driver on the A600+ whether or not an error occurred during the transmission. All error and retry logic was handled within the A600+. Likewise, the software on the A600+ passed outbound messages which were already formatted to the PSI card, which is responsible for the actual transmission to the 3725.

PSI DRIVER

This is a generic driver which we had written for other projects. The driver was designed to reduce to the absolute minimum the number of interrupts needed to transfer messages between the A-600 and the PSI. When the driver receives a write request from a program the entire message is transferred via DMA to the PSI. The A600+ is interrupted only on completion or on an error condition. Messages from the PSI to the A600+ occur asynchronously, so they involve two interrupts of the A600+, one when the PSI requests a transfer, the second when the transfer completes. The PSI-to-1000 interface was also designed to be highly reliable. Data transfer errors are detected and the transfer is retried by the driver. PSI software failures are detected and reported to the pre-driver, in which case the card can be reinitialized.

PRE-DRIVER

The pre-driver has complete control over all the PSI cards on the system. It initializes each card, downloading the protocol code which can be specific for each card. It sends control messages to each line task to configure the line parameters. It communicates with the cards through the PSI driver with class

I/O on two devices, one for reading and one for writing.

FRAME PROCESSOR

In SNA jargon, this module is responsible for the Data Link layer processing. Since we were emulating 600 devices on a single line to the IBM, this line was defined within the IBM as having 20 multidropped 3274 controller units as the Physical Units (PU), each with 30 terminals defined as the Logical Units (LU) attached to it. The frame processor is responsible for checking the PU addresses, maintaining the sequence numbers for each frame, and handling recovery and retries of sequence errors.

SNA PROCESSOR

This was the most complex and time-consuming task to create. The remaining SNA layers, path control, transmission control, session control and data flow control are handled by this task. All messages for a PU and all non-data messages for an LU are processed here. Non-data messages include initialization handling, session parameter processing and flow control. In addition, this task is responsible for maintaining the "state" of the PUs and LUs on the system.

TERMINAL EMULATOR

The Burroughs terminals, like most non-IBM devices, have quite a different keyboard layout than the IBM 3278. We therefore had to implement special keyboard emulation capabilities, such as the Program Function (PF) keys, the Program Attention (PA) keys, the Clear key and the System Request key. Although there are several methods for handling this, our client chose what we believe to be the most straightforward solution with the least impact on existing and newly developed IBM applications.

We arrived at a convention, whereby the user could start any unprotected field on the screen with the "PF notation character" (an exclamation mark was chosen), followed by "Pnn", where nn is the PF key the user wishes to simulate. For example, to simulate hitting the PF10 key on a 3278, the Burroughs user enters "!P10" at the start of any unprotected field on the screen. Likewise, a "!PA2" which starts an unprotected field will cause the terminal emulator in the A600+ to simulate the transmission of the IBM 3278 PA2 key. We were thus able to totally simulate an IBM keyboard without causing the programming staff to redesign every form within the IBM applications system, and without requiring extensive interaction between the user and our terminal emulator.

CHARACTER MAPPING

The Burroughs terminals are capable of handling the standard 3278 set of field attributes (protected/unprotected, highlighted, hidden, numeric/alphanumeric), thus the mapping of control characters from IBM to Burroughs is straightforward. However, because the Burroughs terminals send all unprotected data (whereas IBM terminals only transmit modified fields), the mapping of the entire data streams can be rather complex.

To handle this chore, we defined a 1920 byte array for each terminal into which the IBM data stream was stored on each outbound message. This array (screen

buffer) was coupled with a 1920 bit array which denoted whether or not that character position started a new field. Thus, when the response was received from the terminal, we can "remember" where the unprotected fields resided on the screen, compare the received data with what was originally presented to the terminal in that unprotected field, and determine which fields were modified by the user. In this manner, we are able to build the 3278 data stream which will be transmitted back to the IBM.

FINISHED PRODUCT

As you might expect, the majority of CPU overhead takes place in the character mapping subroutines. Indeed, we have found that, so far, this is the limiting performance aspect of the system. We have found through timing instrumentation to expect that with the A600+ we should be able to handle message rates approaching four messages per second, assuming full, 1920 character messages. The use of a 3-mip A-900 would result in dramatic improvement. The maximum response time under the initial implementation was a disappointing 6 seconds. We knew that the 19.2Kbps line between the Burroughs FEP and our box would add up to 2 seconds to this lag, but certainly did not account for an additional 4 seconds back and forth from the IBM. After some amount of testing, we found the response time problem was caused by the fact that we were emulating 20 3274 controllers on the SDLC line. Even though this is a 56Kbps line, the time involved in polling 20 individual control units, and the time lag before the poll response is received could indeed be responsible for up to 2 seconds additional response time. By changing our emulation scheme from multiple 3274 controllers to the IBM 8100 architecture (which permits 255 LUs per PU), we were able to significantly decrease this latency by decreasing the number of polled LUs from 20 to 3.

ENHANCEMENTS

As mentioned earlier, we are accustomed to moving as much of the line handling and protocol logic as possible out to the PSI card. Since we now know that this card is easily capable of handling 56Kbps links, our next step is to move frame processing out of the A600+ into the card. This will improve the performance and response time of the system for two reasons.

First and foremost, the latency between a poll from the IBM and response from the A600+ will be all but eliminated. Currently, since no frame processing logic takes place on the PSI card, each poll is passed into the A600+ for dissemination, where the frame processor decides the proper response. By placing this logic within the PSI card, this "wait time" is eliminated resulting in faster turnaround. Secondly, by removing frame processing from the A600+, added CPU is available to the other tasks within the system.

Also available, and closer to home for the HP user, is the HP point-to-point protocol and HP terminal emulation capability. Thus the system can serve as a high volume SNA gateway for HP devices in much the same manner as the Burroughs devices discussed in this paper.

Our faith in the HP1000 as the proper computer for communications handling was rewarded. The real time capabilities of RTE and the performance potential of the PSI boards has time and again allowed us to provide an HP solution at sites where, quite often the client has never owned or even considered HP equipment.

When this is coupled with the reliability of the hardware and the professional support of HP field personnel, we feel confident in recommending the A-Series as a communications processor.

1014. INTERFACING AN ARRAY PROCESSOR TO THE HP A-SERIES

Cimarron Boozer
Ninon V. Argoud
Frank M. Phelan

Universal Computing
4710 Ruffner St. Suite A
San Diego CA 92111

The HP A-Series (A600, A700, and A900) has been interfaced to a MAP 300 array processor. The MAP 300 array processor is manufactured by CSP, Inc. (CSPI) and has been manufactured since 1978. The same Universal Computing software and hardware interface works on all three machines in the HP A-Series without modification.

Universal Computing designed the HP A-Series interface on request from a number of companies in the United States and Europe. The A-series interface has successfully been installed on the HP A600, A700 and A900 in the United States, France, and Italy.

PART I: MAP 300 PERFORMANCE AND SELECTION CRITERIA

Our experience with the MAP 300 started in 1979 when it was installed on an HP 1000 F-series computer. Throughout the years, the number of I/O interfaces developed for the array processor and the library of software has grown. For this reason, it was extremely important to have an interface to the new A-series that was compatible with the old system.

Universal Computing has developed a product line for the HP 1000 and MAP 300 combination. Standard products include interfaces to the DATARAM Wide-Word Bulk Memory, LEXIDATA 3400 color display, and RASTER TECHNOLOGIES Model 125 and 180 color display. In addition, numerous interfaces have been designed for customer hardware including high-speed A/D converters and video cameras. For example, 60 frame per second continuous video image processing (using floating point correction) has been achieved at over 2 megabytes per second.

Although there are many array processors on the market today, the MAP 300 is highly suited for many real-time applications commonly found on HP computers. For those unfamiliar with array processors, we have provided the following summary of the MAP 300 architecture, I/O capability, software, and expandability.

MAP 300 Architecture

The MAP 300 is designed to allow blocks of numbers (vectors) to be processed efficiently. The MAP 300 is user programmable and FORTRAN callable. It is capable of speeds in excess of 12 million arithmetic operations per second and runs in parallel with the host computer.

The MAP 300 uses a 25 bit mantissa and a 7 bit exponent for true floating point representation.

The MAP 300 uses a parallel architecture to achieve its processing speed. Two multiplier-adder units operate in parallel with an address processor, a control processor, and one or more I/O processors. Arithmetic routines can thus be written without regard to buffer control or I/O processing.

MAP 300 Input/Output

The MAP 300 offers a separate programmable I/O Processor, allowing the array processor to free itself from the chores of I/O. The MAP 300 can have multiple "I/O scrolls" each providing interrupt driven data streams.

Software

The software used with the MAP 300 includes the host resident support library, the host resident I/O driver, the array processor executive, and the array processor resident arithmetic routines.

The host resident support library is used by most programmers to access the array processor. These routines contain common vector and matrix operations, and are available in the "SNAP" library. Most tasks can be performed using these routines singly or in groups.

The MAP 300 uses an Executive resident in the array processor memory, that schedules tasks, controls data flow, and coordinates activity between the array processor and the host. The Executive and the processing routines (CSPI and user written) are downloaded into memory once. Overhead is minimal because routines can be referred to by 6-word function control blocks (FCB's). FCB's can be grouped together to form function lists, which allow a number of tasks to execute without host intervention.

The MAP 300 array processor utilities include a cross assembler, cross loader, simulator, and diagnostics. Cross assemblers allow programmers to make custom routines for the array processor. The cross loader allows files to be downloaded to the array processor. The simulator program allows the host machine to emulate the array processor. This is useful for verifying complete and accurate operation of user written routines. Array processor diagnostics test both hardware and software.

Expandability

Adding or upgrading memory is trivial on the MAP 300. The memory controller allows any memory configuration on its three busses: three different speed memories can be "mixed or matched". For example, the user can specify high speed "scratch pads" for some operations, while using slower memory for buffering I/O data. Increased processing speed can be accomplished by upgrading from a MAP 300 to a MAP 400. This requires the addition of an extra arithmetic unit, and can effectively double the processing speed to about 24 million floating point operations per second (MFLOPS). Both the memory and the MAP 400 upgrades can be done in the field.

PART II: COMPARISON OF HP A-SERIES TO HP 1000 M/E/F INTERFACES

Hardware Differences

The HP 1000 M/E/F has a special CSPI cable adapter which attaches to one HP 12039A Universal Interface card. For the A-series, Universal Computing developed a special cable adaptor to allow standard CSPI interface cables (VAX-compatable) to attach to two HP 12006A Universal Interface cards. In this way, no special interface was required for the A-series.

Software Differences

DVX74, the driver for the HP 1000, was written in HP assembly language (ASMB). Because most of the work was done by the driver, changes required that a new SYSGEN be performed in pre RTE-6/VM days. In addition to DVX74, a pseudo-driver was written which passed all of its operations via EXEC calls.

The HP A-Series uses the HP supplied driver ID.50. Most of the work is actually done in a FORTRAN written pseudo-driver which handles all bookkeeping operations and does EXEC and custom calls to perform I/O. These custom routines were written in HP MACRO to perform quick I/O operations using the $LIBR and $LIBX library routines. In this way, small (6 word) transfer or control operations could be performed without causing excessive system overhead required by EXEC calls.

The Software libraries and utility programs are identical for both series of computers.

The HP1000 M/E/F is operable with the MAP only under FMGR. The software interface for the HP A-Series was developed to operate under both FMGR and CI.

PART III: A-SERIES INTERFACE CONSIDERATIONS

HARDWARE INSTALLATION

Installing the interface between the A-series and the MAP 300 requires two HP 120006A Parallel Interface Cards (PICS). First, the two PICS require 8 switch selectable options to be set. These switch selectable options are the 6-bit select code (U1S3-U1S8) and the device command sense bit (U1S2). The switches are located near the two cable connectors (J1) on the PIC.

After setting all switches on both PICS, they are installed in two adjacent slots on the HP backplane. The highest priority slot is the data channel and the lowest priority slot is the command channel.

SOFTWARE INSTALLATION

System Generation

An Interface Table Entry (IFT) and a Device Table Entry (DVT) must be added to the System Generation Answer File for each PIC.

The following is an example of adding the IFT and DVT to the System Generation Answer file for PIC #1 and PIC #2. The select codes in the following examples are set to 21 and 26, but can be assigned any valid select code numbers as long as the select codes correspond to the switch settings on the PICS (see Switch Setting). The LU'S are set to 83 and 84 in the example but may be set to any valid LU number as long as they are consecutive numbers and the lowest number is the data channel and the highest number is the command channel.

```
*
* PARALLEL INTERFACE CARD #1
*
IFT,%ID,50::00017,SC:21B
DVT,,,LU:83,TO:5000,DX:2,DP:1:1:2:ST:45B
*
* PARALLEL INTERFACE CARD #2
*
IFT,%ID,50::00017,SC:26B
DVT,,,LU:84,TO:0000,DX:2,DP:1:1:2:ST:45B
```

Loading the Software Using TF

There are three options available to the user when restoring the tape to disc. The user can restore: "N1" installation and runtime software, "N2" SNAP executive source listings and hex object files, or "N3" development software.

LU Configuration for the MAP 300

Routine IMAPI has an option which returns the MAP LU. After the user has finished the system generation and restored the software to disk, the user must edit the file &IMAPI and change the MAP LU to correspond to the system generation. After the change is made, the routine will be located in a library which all users can search.

Routine IMAPI can also be called with an option which allows the user to access multiple MAP 300 array processors on the same HP system.

If the user wishes to lock LU's other than the MAP using HP LURQ calls, the user must first call IMAPI to get the MAP LUS and include them in the list of LUS to be locked. This call to IMAPI will inhibit the MAP driver from calling LURQ.

DOWNLOADING SOFTWARE TO THE MAP

In order to use the array processor, a software executive must be loaded into its memory. The utility program MPLD is used to perform this function. There are two types of executives: (1) the diagnostic executives, and (2) the SNAP II executive. The diagnostic executives are contained within each of the host loadable diagnostic files. In order to simplify the loading of the SNAP executive a command procedure called *EXEC is provided. To load the SNAP II executive the user issues the following command:

```
CI> TR,*EXEC
```

Also supplied is MPLD, a program which can: (1) Transfer an object file to the

MAP, (2) Transfer a binary file to the MAP, or (3) Convert a object file to a binary file.

After loading the SNAP executive and linking with the SNAP library ($SNPLB), the user program is ready to run.

REFERENCES

[1] HP 12006A Parallel Interface Reference Manual (Manual Part No. 12006-90001) January 1984

[2] Growth Pains of A Shipboard HP-1000: From Paper Tape to Array Processor (HP1000 International User Group Conference, Exploitation '83) p.37

[3] MAP 300 to HP A-Series Installation and Operating Manual (May 1985), Universal Computing

1015. CONTINUOUS AIR MONITORING USING AN HP1000-CONTROLLED DATA ACQUISITION SYSTEM

William C. Miller, P.E.
City of Philadelphia
Department of Public Health
Air Management Services Laboratory
1501 E. Lycoming Street
Philadelphia, PA19124

INTRODUCTION

The Air Management Services Laboratory (AMS) is responsible for operating a city-wide air pollutant monitoring network as required by federal EPA regulations, state law, and city ordinance. This monitoring effort is intended to report actual air quality with respect to established health related and other national guidelines. Short-term (daily) information is reported to the public in a manner known generally as an 'air pollution index.' Long-term study of the air quality data is done by both the EPA and AMS in order to determine general progress in attainment and maintenance of federally mandated air quality goals. This process permits periodic adjustments to federal and local enforcement policy based on actual air quality trends.

Air quality data is a highly visible indicator of an air pollution control agency's effectiveness in protecting the air environment. The system used to measure and collect this data must, therefore, be highly reliable, timely, and possible to operate within the expected budget constraints of a municipal government.

AMS currently operates thirteen continuous air monitoring stations located throughout the city. Pollutants measured at these sites include sulfur dioxide, carbon monoxide, nitrogen oxides, ozone, hydrocarbons, particulate matter, and lead. A variety of instruments are used to continuously analyze the air sampled at these remote sites. With the exception of lead, the result of the analysis is a voltage level which indicates the concentration of the pollutant in the atmosphere. This voltage must be sampled, recorded and eventually converted into the pollutant concentration.

In the original monitoring system operated from 1965 through 1972 the voltages were recorded on a continuous strip chart and later evaluated by a technician who estimated the hourly average concentration from the trace. This approach becomes very labor intensive with even a modest monitoring effort and provides no real-time information or data quality assurance capability. In the early 1970s AMS was required by the newly implemented Clean Air Act to expand its original two station, manual air monitoring network to an eight station, automated system. This project involved the installation of six new remote sites, an analog telemetry system and an HP model 2100S central computer system. Software development for this system was done by AMS engineering and technical personnel. The system went into service in the fall of 1974.

After several years of operation it was determined that the analog telemetry system was not performing adequately. In 1978 AMS initiated a major upgrade of

the system which included the installation of an HP-1000 E-series CPU and the development of a digital telemetry system and associated quality assurance system to replace the analog system. AMS contracted with the Wallace-Fisher Instrument Company (WF) for the design and construction of the system electronic hardware. Revision of the system operating software was done primarily by AMS personnel.

SYSTEM HARDWARE CONFIGURATION

The AMS Laboratory Computer Center currently operates an HP-1000 system based on an early E-series CPU (HP 2113B) with one megabyte of memory operating under RTE-6/VM and Session Monitor. Major peripheral subsystems include an HP 7925 disk drive, an HP 7914ST mass storage system consisting of an HP 7974 dual-density tape drive and an HP 7914 disk drive, an HP 7970B tape drive, an HP 2563A system printer, a graphics subsystem including an HP 2647A terminal and an HP 9872A plotter, two eight-channel multiplexors, ten terminals (hard-wired or dial-up), and two HP 150C MAX personal computers. The computer center supports general agency data processing needs in program development and database management.

Figure 1 shows the major components of the real-time data acquisition system. Communication between the HP-1000 and the Wallace-Fisher PH-16 telemetry system is handled through a standard HP 12966A interface card. The AMS implementation of the PH-16 requires the use of a second I/O slot for a relay control card which is used to control the central modem and telephone relay rack. Each remote telemetry set (RTS) is connected to the central relay control rack by standard quality, dedicated, two-wire telephone circuits. The maximum number of remote sites is sixteen with each RTS capable of reporting up to sixteen voltage inputs.

The PH-16 system communicates using RS-232C protocol at speeds up to 60 characters/second. Each RTS is hard-coded with a unique two-character ASCII address so that it may respond properly to the system polling software. This feature is most important in the alternate configuration having no central relay control with all RTS sites 'looped' in one continuous telephone circuit. Each input channel of the RTS has an individual input amplifier so that the monitoring instrument voltage level can be standardized to a specified voltage range. These analog voltages are digitized to 1/2000th of the full scale value and transmitted to the central computer as ASCII characters.

A major feature of the PH-16 system is the ability to send and receive command and status information as part of the polling and response messages. Command bits sent to the RTS can be asserted as low level digital signals, relay contact closures, or to operate a digital-to-analog output circuit card. The response from the RTS can report the status of implemented digital inputs to report a variety of conditions such as relay closure, alarms, temperature sensors, and the 'on-line' status of each input channel amplifier.

The PH-16 can be configured in a truly remote manner by installing a data-logging 9-track magnetic tape unit in place of the normal telephone communication hardware. The format of the data recorded in this manner is consistent with that returned by the telemetry to the central computer.

DATA ACQUISITION SOFTWARE

The air quality data system operating software consists of three major components: data acquisition, quality assurance, and reporting. The fundamental data item derived from this system is an hourly average pollutant concentration. The concentration is usually expressed in units of parts per million or micrograms per cubic meter. Long term averages are calculated from the hourly average data. Federal air quality standards for each pollutant are commonly expressed in several sample averaging periods. For example, the standards for carbon monoxide are specified as a one hour average, and eight hour average, and an annual average.

Primary system data files include the air quality data file (AQDATA), the status and command bit definition file (STATUS), and the input channel definition file (CHAN). Other related files contain definitions of the monitoring instrument characteristics, a description of each monitoring site, and error logging data.

The AQDATA file is a direct access file with each record containing the hourly average pollutant values for each instrument reporting. This file is presently capable of holding 100 days of data. Each new hourly record overwrites the oldest hour on file. After appropriate quality assurance this 'on-line' data is archived in monthly files on magnetic tape.

The STATUS file contains definitions and message strings for the 56 possible status bits for each PH-16 RTS. The CHAN file describes each of the 256 possible telemetry input channels including the type of pollutant measured, the analytical method used, EPA codes, equation coefficients for converting voltages into concentrations, and equipment ID numbers. It is also possible to define derived measurements in the CHAN file. This feature is used to calculate the mean value and standard deviation from measured wind speed and direction. These statistical values are then stored in AQDATA as hourly data.

Both STATUS and CHAN are ASCII files. Information in these files is routinely edited to reflect operational changes or adjustments to conversion equation coefficients. Utility programs are used to reduce both files to bit maps for rapid access by the real-time data acquisition system. These and other reference files and tables are stored in a small global scratch file area located on the auxiliary subchannel (LU 3). This global area is used to share information and for short-term data storage as required by the system software. Track and sector addresses for the various global files are kept in a simple directory table in a system subroutine. Files are referenced by code names which are converted into addresses for the appropriate EXEC read and write calls. For example, the name of the status bit map is 'ST' and the telemetry channel definition map is called 'MP'.

The major software components of the data acquisition system are shown in Figure 2. The system initialization program START sets the system time and schedules the SET75 and MINIT programs at system boot-up. Sampling of the RTS voltages is scheduled once a minute. These minute samples are averaged by the HRAVG program at the end of the hour. The results are then stored in the AQDATA file.

The RTS polling and control program SET75 executes at 7.5 seconds after the minute. The basic functions of this program are the operation of the telephone

line switching relay, sending the polling message to the RTS through the BACI card, reading the response message from the RTS and logging error conditions. Operation of the interface card is managed by the DVR75 driver program developed by AMS for this application.

The polling message is constructed by SET75 for each RTS. The message consists of ten ASCII characters: the first character is a constant indicating the start of the message; the next two are the RTS 'name'; six characters follow carrying 7 bits of command data each; the final character is another constant which signals the RTS to scan the 16 input amplifiers and process commands. The RTS responds with a 112 character message consisting of seven characters for each input channel. The seven characters include: one character to indicate the channel number; three ASCII numbers representing the decimal digits from the DVM; one character containing bit status information concerning the polarity, range, and status of the DVM value; one character carrying status bit data from the RTS command components; a final character serves as a delimiter. The RTS names and command values required for the SET75 polling message and the data returned to SET75 from the RTS are stored in system common.

The MINIT program is scheduled to execute near the end of the minute to capture the digital voltage values and to process the return status bits. Each minute's data is moved to a file in the global scratch area. At the end of minute 59 the MINIT program schedules the HRAVG program to convert the minute data into hourly averages and to report messages to the system console as required by the status bit definitions.

Cooperation among the various programs involved in this process is managed by the use of a global resource number stored in system common. Other flags in system common indicate the on-line status of each RTS input channel so that empty channels are not processed. Error conditions and 'bad scans' are reported by SET75 to an error logging program which saves the RTS response in a global area file for diagnosis by engineering personnel to determine the source of telemetry system problems.

QUALITY ASSURANCE AND REPORTING

An integral part of the air monitoring network operations is the use of quality assurance procedures to establish a statistical data base to indicate the precision and accuracy of the analytical instruments. This statistical data is required for all instruments designated as reporting data for the EPA mandated portion of the network. At present there are 24 instruments in this category.

Instrument precision checking basically involves challenging the instrument with a known concentration of the pollutant at the remote site and observing the MINIT program results with a portable terminal. This procedure allows the technician doing the procedure to see immediate results. Adjustment to the voltage equation conversion coefficients in the CHAN file may be made to compensate for minor differences between the input and output concentration values. Extreme disagreement between the values requires additional procedures to correct the error.

A fully automated precision check procedure can be accomplished by using the

command capability of the PH-16. By providing appropriate remote site equipment (gas tanks, pumps, flow controllers, valves, etc.) the automatic procedure and the recording of concentration values can be initiated and managed by the CALIB program. This program periodically scans a master file containing general commands such as procedure number, starting time, and site-instrument matrix coordinates. After CALIB schedules itself to execute, the required procedure command file is interpreted and implemented. The operation of remote quality assurance equipment is controlled by CALIB through the setting of command bits in system common. The CALIB procedure commands are sent by SET75 to the remote site and obeyed as required. The status bit response from the RTS can be reviewed by CALIB with adjustments being made to the procedure as necessary. Minute data values and other messages are recorded in the corresponding 'result' file for later statistical analysis.

Data reporting and archiving procedures are done on a monthly basis. The hourly records for one month are copied from the AQDATA file to magnetic tape. The records for one month are also reformatted into EPA SAROAD format records and copied to magnetic tape. The EPA data is submitted quarterly with the required quality assurance supporting data. A daily report of the Pollutant Standards Index (PSI) is prepared from the AQDATA file and distributed as public information.

SYSTEM PORTABILITY

In 1980 the City of Houston contracted with Wallace-Fisher for the installation of the PH-16 telemetry and an HP-1000 computer system to operate an existing air monitoring network. The modular nature of the AMS system software and the extensive use of tables and reference files made it fairly easy to transport the system to Houston. AMS personnel assisted with installation, testing and training for the Houston version of the software. In an informal sense the City of Philadelphia acted as a free consultant to the City of Houston since it was viewed as an advantage to have another user of the PH-16 in an identical application. The major difference between the two systems is the use of the non-relay, looped telephone circuit arrangement in Houston. This is appropriate since there are about half as many remote sites involved. It is expected that the PH-16 system could be applied to a variety of similar applications such as radiation monitoring, storm sewer flow control, and meteorological monitoring.

COMMENTS

The most reliable component of the air monitoring system is the HP-1000. The total down-time for the E-series CPU and major peripherals due to equipment failure since going on-line in 1978 is less than a week. Some mechanical problems with the PH-16 relay contacts occurred last year but were solved with the proper cleaning solution. The modular construction of the PH-16 hardware makes troubleshooting and repair fairly easy. The majority of data loss is attributed to telephone circuit interruptions and power failures.

The best feature by far is the command capability which allows the automation of the quality assurance procedures. A manual precision check procedure requires the time of one person for most of one day for each instrument. This time includes travel to the remote site, equipment set-up, and the domination of a

dial-up terminal port. The continued development and application of these automated procedures has top priority at AMS.

ACKNOWLEDGEMENTS

The development and operation of the AMS air monitoring network is supported in part through a program grant from the U.S. Environmental Protection Agency under Section 105 of the Clean Air Act. I would like to express my appreciation to Stephen Fisher and to the employees of Air Management Services for their efforts in providing a clean air environment for the citizens of Philadelphia.

FIGURE 1. Hardware Configuration

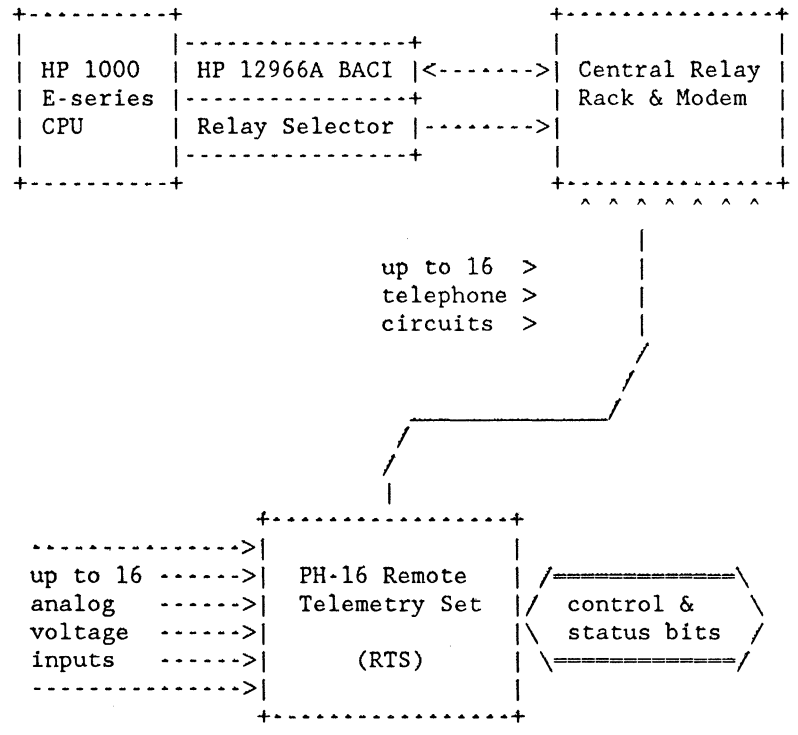
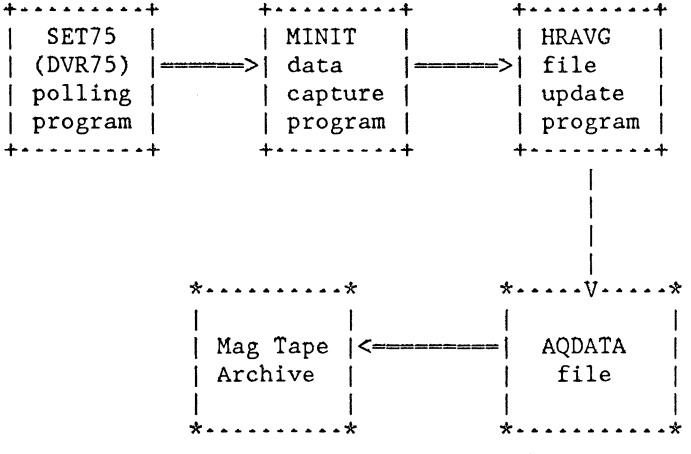


FIGURE 2. Software & Data Flow



1016. INTERACTIVE PROGRAMMING

Donald A. Wright
Interactive Computer Technology
2069 Lake Elmo Avenue North
Lake Elmo, MN 55042 USA

INTRODUCTION

If there were a Bill of Rights for computer users and operators, it would include, among other things, a list of features to be expected in all software packages to make them easy for anyone to use. These features might include: automatic casefolding (now available in most HP programs), automatic justification (deletion of leading spaces), free exit (the ability to exit from the program at any point), delimiting parameters on commas or spaces, and much more. Even without the Bill of Rights, a user certainly does have the right to expect to find such features in software written especially for his/her own application.

This paper describes four types of programs employing differing methods for operator interaction. Techniques appropriate to each program type are explored in some detail, with emphasis on the newest RTE Library Subroutines and some new CSL contributions. While the basic principles discussed are of value in all languages, the examples are in Fortran 77.

PROGRAMS USING RUN-STRING PARAMETERS

When a program uses no operator inputs other than runstring parameters, it may stretch credibility a bit to classify it as an interactive program. Nonetheless, this is a very common type of program, and several things can be done to make such programs easier to write and to use.

*** RUN-String Parameter Recovery:**

Run string parameters (and, in fact, most other commands and parameters) are easiest to deal with if they are first recovered into the program in their original form - as ASCII character strings. The least-well-documented parameter-recovery subroutine in the Fortran 77 Manual is the easiest to use: FPARM. A typical FPARM call is as follows:

```
CHARACTER INPUT1*20, INPUT2*20, - -
-
CALL FPARM (INPUT1, INPUT2, - -)
```

FPARM accepts any number of input parameter strings of any length, and will place the separated values passed by the RUN command into these strings, blank-filling the excess length of any string which

receives a parameter. When a value is not supplied in the RUN command, the corresponding string is unchanged (NOT blank-filled).

Once received into the program as character values, numeric runstring parameters (if any) can easily be converted to the appropriate numeric type by using a Fortran READ or by using one of the new RTE Relocatable Library functions such as DECIMALTOINT. This is further described under the heading PARSING.

* Usage Help:

A very simple technique applicable to programs which accept no interactive input other than runstring values is the Usage display. It is displayed when the program determines that at least one required parameter has not been supplied at all. For example, the LI program requires that the operator supply at least the filedescriptor of the file to be listed. A Usage display for LI might be coded this way:

```
CHARACTER FILENAME*64, OPT*1, FIRSTLINE*12, LASTLINE*12

FILENAME  = ' '     ! Make sure the variables are
OPT       = ' '     !  initially blank, not null
FIRSTLINE = ' '
LASTLINE  = ' '
CALL FPARM (FILENAME, OPT, FIRSTLINE, LASTLINE)
IF (FILENAME .EQ. ' ') THEN ! Oops - no file name
  WRITE (1, *) 'Usage: LI file [opt] [range]'
  GO TO 9999                ! Exit from program
ENDIF
```

This Usage display requires just 4 lines of code (the IF-THEN-ENDIF block). But such a simple addition to a program can be an enormous help to an operator, often eliminating the need to spend valuable minutes consulting a manual. Of course, a Usage display can be much longer than a single line if called for by the application. Many of the newest HP utility programs have a Usage display.

SCREEN-ORIENTED PROGRAMS

Screen-oriented programs (forms, screen-driven programs) can be developed from scratch, but are MUCH easier to develop using any one of several different purchased packages. Some vendors of such software known to this author are: C & L Systems, CCS, HP, and Polaris (alphabetical order).

Disadvantages of screen-oriented operator interaction are:

* Initial cost.
* Only HP terminals can be used.
* Time is required for screen changes.

Advantages are:

* Much easier for an inexperienced user.
* Fast if there are many prompt fields per screen.
* Packages can provide many tools which make interactive programming much easier.

Techniques for making the best use of screen-oriented programs differ considerably with the forms package purchased. But some of the basic principles are the same for screen-oriented programs as for all others. Even screen-oriented programs should provide at least these functions:

* Free exit from the program at any time (unless exit from the program is not to be allowed at all).
* Ability to restart the program at any time.
* Temporary escape to other programs.
* Appropriate handling of illegal operator entries.
* Easily available help facilities.
* Casefolding & justification of selected input fields.

All screen-mode packages permit the application of these principles, but they must still be implemented by the programmer.

PROGRAMS USING SEQUENTIAL PROMPTS AND RESPONSES:

This is perhaps the most common type of program written for very specific applications, especially when the program is intended to execute in the same sequence (if not with the same data) every time. The program prompts for one specific input (parameter, variable, whatever), then another, etc. until it has sufficient information to execute to completion.

Examples of this program type are the RTE-6 Generator RT6GN and the SWTCH program. The responses to the prompts in RT6GN are nearly always supplied from a file rather than interactively, but operation is the same. When an error occurs, RT6GN reverts back to interactive prompt and response. Some of the following principles of interactive programming do appear in these HP programs, and a few do not:

* Casefolding:

Two RTE Library subroutines now make casefolding (upshifting of lower-case letters a-z) very simple indeed:

CALL CASEFOLD (STRING) for character variables.

CALL CLCUC (ISTRING, LENGTH) for strings contained in integer arrays.

New programs written in Fortran 77 will obtain the most flexibility by keeping operator inputs in character variables and will use the

CASEFOLD subroutine. Older programs which already keep operator inputs in integer arrays can be easily upgraded with CLCUC. Both subroutines have exactly the same function: the characters a-z are changed to A-Z, and no other changes are made in the string at all.

*** Left-Justification:**

How do blank characters appear at the beginning of an operator's input? Who knows. But we all do it from time to time, and when such an input line is processed the leading blanks often must be deleted to prevent errors which will be unclear to the operator. If operator responses are being read from a file it may be even more important to delete leading blanks which may be put there for appearance reasons. A contributed CSL subroutine called LEFTJUSTIFY is the easiest to use:

```
CALL LEFTJUSTIFY (STRING)  for a character variable STRING
```

The HP subroutine SPLITSTRING will left-justify a parameter as it is being split off from a string, but it will not leftjustify an entire string containing delimiters (such as blanks) nor will it leftjustify a string in place.

*** Free Exit:**

This is a MOST important principle in the user's Bill of Rights. No matter what dumb thing I may have done in a program I must be able to exit at virtually any point without penalty (unless I have reached a point where files are modified and it is just too late). This is easily implemented by checking for a specific EXIT command at every operator input. 'EX' is the one most commonly used in HP programs, and is recommended. When the EX is detected, the program should close files and perform any other necessary cleanup, then terminate.

*** Operator-Input Subroutine:**

Since every interactive prompt and response requires some common actions, these can often be combined in a simple input-processing subroutine such as the following:

```
   SUBROUTINE INPUT (STRING, *)
   CHARACTER*(*) STRING
   READ (1, '(A)', END=5) STRING  ! No runtime errors
   CALL CASEFOLD (STRING)         ! Upshift a-z to A-Z
   CALL LEFTJUSTIFY (STRING)      ! Delete leading blanks
   IF (STRING .EQ. 'EX') RETURN 1 ! Alternate EX return
 5 RETURN                         ! Usual return
   END
```

With such a subroutine the prompt-response portions of a program may be coded as follows:

```
   WRITE (1, *) 'What is the title? _'
   CALL INPUT (TITLE, *9900)       ! 9900 is exit
```

If the expected input is numeric, further processing of the input character string is required. This can be done with the Fortran formatter or by using the new RTE Library numeric conversion subroutines. This example provides free exit, numeric conversion, and runtime error traps:

```
 4 WRITE (1, *) 'What is the pressure (PSI)? _'
   CALL INPUT (STRING, *9900)
   READ (STRING, *, ERR=4, IOSTAT=IOS) PRESSURE
```

*** Handling Illegal Operator Responses:**

The preceding example shows how input strings containing illegal numeric responses can be handled. Nothing will bring a program user back to the programmer faster than a message like

```
     *RUNTIME ERROR* 0494 @ 02016.
```

Trapping and handling such cryptic messages is one of the biggest favors a programmer can offer to both parties.

In many instances it may also be helpful to test the received value for a valid range, as in this example:

```
   IF (PRESSURE.LT.13.6 .OR. PRESSURE.GT.33.3) THEN
      WRITE (1, *) 'Pressure must be between 13.6 and 33.3'
      GO TO 4
   ENDIF
```

Such a test can often prevent the operator from unwittingly entering an inappropriate value which might alter the program's results in a confusing way.

The most important technique in dealing with illegal responses is to repeat the prompt, as in the above examples. In some cases it may also be desirable to reduce confusion by explaining why the prompt is being repeated, as in the example above.

*** Input from Files:**

When a program is capable of accepting a predetermined sequence of responses from a keyboard, it is usually a simple conversion to make it accept its inputs from any arbitrary device or file. Usually the input device/file is specified in the run string, defaulting to the interactive terminal if not specified. Whatever device or file is specified in the following example will be referred to as Fortran Unit 1 after being opened:

```
   CHARACTER*64 FILEDESCRIPTOR
   LOGICAL*2    INTERACTIVE
```

```
.
   FILEDESCRIPTOR = '1'                ! Set default
   CALL FPARM (FILEDESCRIPTOR)         ! Device/file name
   INTERACTIVE=FILEDESCRIPTOR.EQ.'1' ! Set flag for later
   OPEN (1, FILE=FILEDESCRIPTOR, IOSTAT=IOS, ERR=2,
  & STATUS='OLD')
 2 IF (IOS.NE.0) THEN                  ! Oops - OPEN error
      Deal with the error
   ENDIF
```

The interactive prompt-response sequences then should be altered to inhibit display of the prompts except when Unit 1 is the interactive terminal. Typical prompt-response code might be:

```
   IF (INTERACTIVE) WRITE (1, *) 'What is the Title? _'
   CALL INPUT (TITLE, *9900)       ! 9900 is exit
```

It may be even more efficient to design the INPUT subroutine so that it issues the prompt in addition to collecting the response, and so that it is capable of inhibiting the prompt when input is not interactive.

*** Restart:**

In some cases it may be very desirable to permit the operator to restart interactive inputs from the very beginning, or to restart a particular portion of them. This can be accomplished easily by adding an alternate return to the INPUT subroutine and testing there for a command such as "/R", or by using Control-D as the Restart command as follows (Control-D will cause the READ statement to go to the statement specified by END=):

```
   READ (1, '(A)', END=1) STRING
```

PROGRAMS WITH A MENU AND COMMAND STRUCTURE:

Many programs have a standard prompt and can accept a variety of different commands at that prompt. Examples are FMGR, CI, EDIT, and very many more. This structure allows the user (and the programmer) the most flexibility, but of course it is somewhat more complex than the Sequential Prompt and Response structure.

All of the principles discussed for Sequential Prompt and Response programs apply here, and more. A program of this type is usually designed to return to a particular point in the code after completing each command. This means that the processing of operator (or TRansfer file) input occurs at only one point and need not be moved outboard to an external subroutine.

A simple prompt with minimum processing of the response might be as follows:

```
   IF (INTERACTIVE) WRITE (1, *) 'Entry> _'
   READ (1, '(A)', END=10) STRING
   CALL CASEFOLD (STRING)
   CALL LEFTJUSTIFY (STRING)
```

*** Command Stack:**

At this point a command stack is useful in menu-driven programs. A contributed CSL subroutine named CMDSTACK is called as follows:

```
   IF (INTERACTIVE) CALL CMDSTACK (LOG, STRING, STACK)
```

LOG is the LU of the interactive terminal (usually 1), STRING is the just-entered string (before or after CASEFOLD and LEFTJUSTIFY), and STACK is a CHARACTER variable in which the CMDSTACK subroutine maintains the stack entries. STACK is typically sized to at least 256 characters. There is no limit on the number of entries that CMDSTACK can maintain in STACK; the limit is on the total number of characters. New entries will push out the oldest ones to make room if necessary.

CMDSTACK expects the same slash (/) commands as are used by the HP programs CI and EDIT. If the entered string is not such a command it is copied into the stack and returned in STRING unchanged except that it will be left-justified. If the string is a stack command (/), CMDSTACK will display the stack and accept an edited input just as CI and EDIT do.

CMDSTACK adds just over 400 words to the program size, plus the size of the STACK variable (typically 128 words or more). It is written in MACRO and is fully documented in its source in the CSL Library (most-recent issue).

*** Command Parsing:**

A command may consist of the command characters alone (e.g. 'EX'), or it may contain one or more qualifying parameters (e.g. CO FILE1 FILE1). After the command is received and initial processing is complete, parameters need to be separated off and their numeric values resolved (if any). The following code is a fairly complete input-processing sequence for a 2-character command and two possible parameters. The SPLITSTRING subroutine is part of the RTE Relocatable Library; it divides the string at a comma or at one or more spaces:

```
   CHARACTER CMD*3,                    ! Command (2 char + space)
   CHARACTER PRAM1*78, PRAM2*78        ! String parameters
   INTEGER*2 INT1, INT2                ! Integer parameters
   LOGICAL*2 ERR1, ERR2                ! Integers valid?
   INTEGER*2 DECIMALTOINT              ! RTE number conversion
   .
   .
 2 IF (INTERACTIVE) WRITE (1, *) 'Entry> _'
   READ (1, '(A)', END=10) STRING
```

```
IF (INTERACTIVE) CALL CMDSTACK (LOG, STRING, STACK)
CALL CASEFOLD (STRING)

CALL SPLITSTRING (STRING, CMD,   STRING) ! Split out
CALL SPLITSTRING (STRING, PRAM1, STRING)
CALL SPLITSTRING (STRING, PRAM2, STRING)

INT1 = DECIMALTOINT (PRAM1, ERR1)        ! Convert
INT2 = DECIMALTOINT (PRAM2, ERR2)
```

The above code can be used with all entered strings with no danger of runtime errors. Error testing is then done after branching to the specific module or code section where the command is processed, since each command will have different requirements.

*** Command Branching:**

When a command has been separated from the entered string, then the program can branch to the section where that command is to be handled. The simplest way to branch (not the most efficient) is to use a list of IF's. The following list makes easy code reading:

```
IF (CMD .EQ. '  ') GO TO 2    ! Nul entry
IF (CMD .EQ. '??') GO TO 6000 ! Help
IF (CMD .EQ. '? ') GO TO 6000 ! Help again
IF (CMD .EQ. 'HE') GO TO 6000 ! And again
IF (CMD .EQ. 'EX') GO TO 9900 ! Exit
IF (CMD .EQ. '/R') GO TO 10   ! Restart
IF (CMD .EQ. 'TR') GO TO 5000 ! TR file command
-
-
```

A somewhat more efficient way to code the branching is to establish a list of possible commands in a character array, use a DO loop or the Fortran INDEX function to find the ordinal position of the command in the array, and use that ordinal position value in a Computed GOTO. In the HP/1000, the computed GOTO is microcoded and occupies VERY little code space.

*** Running Programs (Temporary Escape):**

When an entered command is not found in the command list or the list of logical IF's, then execution will fall through to the next executable code. At this point it is convenient to attempt to execute the command as if it were a RUN (or implied RUN) command rather than local. A new RTE Library subroutine FMPRUNPROGRAM is very slick in this application. In fact, it is the very same subroutine used by CI to schedule other programs. It is called as follows:

```
IERR = FMPRUNPROGRAM (STRING, LRMPAR, RUNNAME)
```

STRING is a copy of the original keyboard-entered string, LRMPAR is a 5-word integer array to receive the RMPAR parameters back from the scheduled program when it completes, and RUNNAME returns the true name of the scheduled program. IERR returns an error code if a problem was encountered (such as -6, file not found).

If the string passed to FMPRUNPROGRAM does not begin with 'RU' or 'XQ', then FMPRUNPROGRAM assumes 'RU'. Blanks or strings of blanks in STRING are treated as single commas.

The unfortunate penalty of FMPRUNSTRING is that it adds about 6 pages of code, with all of the subroutines it pulls in. This is a maximum value - it will be less in some applications because some of those subroutines will be required anyway by other parts of the program.

* Help:

There are at least three good ways to implement a help function. The simplest is for the program itself to contain WRITE statements and FORMAT statements (or character strings) to display help for each requested help parameter. The problem with this solution is the excessive program space required for large FORMAT statements, so it may be appropriate only for modest Help displays.

A second method employs the FMPLIST subroutine. It lists a specified file to the interactive terminal, and can be directed to list only specific lines so that 'subfiles' can be listed if needed. FMPLIST imposes a substantial size penalty on the program, and is not fast when the lines to be listed are not at the beginning of the file.

The third method is the contributed CSL subroutine LIST and its companion program ADDIX. A keyworded file (where keywords are identified in advance) is first created using EDIT. Then ADDIX is run to add an index (to the keywords) on the end of the original file. When LIST is called by the program using it for the HELP function, it opens the ADDIX'd file and performs a direct-access binary search on the index to find the requested keyword, then begins the listing directly at that keyword.

The file which ADDIX uses for input may also be the source for the program's manual, if one is to be provided. It contains forms-control characters in column 1 to facilitate printing of the manual, and those same characters can be used by ADDIX in the identification of keywords.

LIST adds 600 words of code directly to the program, and several pages more if none of the usual FMP subroutines are called elsewhere in the program. But it is fast: In a recent test of speed on a 3100-line Help file containing 212 keywords, the longest time required to begin displaying text was 210 milliseconds, the shortest 50, using a 1-block DCB for the file. This test was performed on a Micro-600 system using the Integral Disk, which is substantially

slower than most CS/80 or MAC disks. The speed is possible because the index can be randomly accessed for the binary search, and because each index entry points directly to the word position of the keyword it represents, again permitting direct access.

LIST and ADDIX comprise the HELP facility used in the commercially available CONNECT terminal-emulator package. They have been contributed to the HP/1000 CSL and are fully documented in their source files.

REFERENCES:

1. RTE-A Programmer's Reference Manual.

2. Relocatable Libraries Programmer's Reference Manual.

3. INTEREX HP/1000 CSL.

1017. USING THE HP-1000 TO AUTOMATE A MACHINING AND GAUGING PROCESS

C. Pappagianis
RCA/Missile and Surface Radar Division
Moorestown, New Jersey 08057

BACKGROUND

The objective of this project was to improve and automate the machining process for a circuit component made of a plastic material subject to unpredictable spring-back. The original process used templates, a manually controlled milling machine, and hand-gauging to determine if additional machining was required to meet the depth specification. The large number of machining positions possible for each piece/pattern made this a time-consuming, error-prone, and costly operation unsuitable for full-scale production.

We started by searching the market for an automated machine capable of communicating with our HP-1000, which contained a database of part locations and dimensions to be machined. In addition to accepting coordinate data over an RS-232 line, the candidate controller had to have a high degree of repeatability and accuracy in positioning the work pieces.

APPROACH

The Automation Unlimited Work Station with the C60 Motion Controller was chosen because it met these requirements and was available "off the shelf." The work station is a large, X-Y table driven by stepper motors. It operates like a large, flatbed plotter--the Y-motion of the table corresponds to the paper motion of the plotter, and the X-motion of the tool arm, which is mounted on a beam above the table, is similar to plotter pen motion. Mounted next to the tool arm is a probe-like linear transducer that serves as a gauge. Both the tool arm and the gauge are capable of independent vertical motion.

The C60 Motion Controller is a Z80-based microcomputer with a plasma display for all position information, and a membrane keyboard for programming and the manual operation of the table. The C60 can also be programmed through an RS-232 serial port; a program can be written on a small computer and downloaded to the C60 for execution.

Programming the C60 is very simple. A program consists of the position data, which implies a motion to that X-Y location, and a tool command in the form of a four-digit number from 0000 to 6999. The tool command actually determines what will happen when the table reaches the specified position; the command options range from "no operation" to the calling of subprograms and the sending of characters via the communications port.

Although the C60 has a built-in microprocessor, there were several application-specific reasons why it had to be interfaced to the HP-1000. The C60 was originally designed to handle repetitive tasks. An operator writes a program to perform each task, be it simple or complex, and either keys it in from the front panel or downloads it from the text editor of a small personal computer (PC). However, data cannot be passed to subroutines in the C60, and its decision-making capability is very limited. In our production process application, the system must be

capable of handling many different parts, each demanding unique machining operations. Writing a separate program for each part is therefore impractical. Furthermore, the coordinate and dimension data for each part is generated on the HP-1000 and maintained in a database. Finally, the C60 contains no provision for reading the post-machining gauge. The resilient nature of the material being machined required this additional gauging operation; an HPIB DVM was therefore added to read the gauge and feed the information back to the HP-1000.

THE INTERFACING PROCESS

Interfacing the C60 to the HP-1000 presented a few problems. The C60 uses a strictly hardwired handshake. In addition, the ENQ and ACK characters have special meanings to the C60. The use of an HP-12966 BACI interface together with a special driver (e.g., a DVF00 or DVW00) was considered, but the BACI card would have used a slot in an already crowded card cage. We opted for the versatile HP-12792B 8-channel multiplexer (MUX), which was available with spare ports. The MUX turned out to be a good choice, since the system configuration had to be changed several times before we found one that worked reliably.

The communication parameters on the C60 are not as flexible as those on the MUX, so the MUX was reprogrammed to conform to the data format the C60 expected, i.e.:

* 7 bit ASCII;
* 1 start & 2 stop bits;
* even parity;
* 2400 baud;
* no ENQ/ACK handshaking.

This was easily accomplished by passing the correct word value to control function code 30B for the MUX driver.

PROGRAMMING THE SYSTEM

Programming the C60 is simple; a single ASCII character emulates each of the front panel keys. Control and alpha characters represent commands; numerics are automatically considered data. There are two ways to execute commands. The first is a fully programmed method: the entire program is entered and the C60 is instructed to start execution. The other method is to put the C60 into Manual Data Input (MDI) mode, where each command is executed in real-time as it is received. Both methods were used in this project.

The heart of the HP-1000 program is two subroutines that send data to the C60. Each makes use of a C60 feature that allows transmission of a character to a remote computer; we used this feature as a substitute for handshaking. Overrun is not a problem, as the subroutines always send data in short bursts and the C60 has adequate on-board buffer.

The first subroutine handles coarse movement around the table in MDI mode. It sends a single set of coordinates to the C60 together with a motion command, and, immediately after, another command that causes the C60 to transmit a carriage return. Then the subroutine establishes a pending read on the C60 LU. The C60 executes the commands serially; it does not pull the next command from its buffer until execution of the previous command has been completed. In other words,

the HP-1000 is "paced" -- it must wait until the pending read is satisfied.

The second subroutine handles the step-and-repeat portion of the machining process. It constructs a complete C60 program "on the fly" based on the dimension data stored by another program on the HP-1000. This includes (1) moving the tool into position; (2) doing the machining; (3) moving the tool away; (4) moving the gauge into place; and finally (5) sending the pace character. Once the gauge is in place, the HP-1000 takes a reading on the DVM via the HPIB and decides if the piece has been properly machined; if so, the system returns to MDI mode and moves to the next location, and if not, the program constructs a new C60 program that increases the depth setting of the machining tool to compensate for the spring-back. The process is then repeated. To keep the program from getting "hung up", a maximum of five attempts to machine a single (stubborn) location is allowed before the program moves the tool to the next machining site.

CONCLUSION

By designing and programming an automated machining system, we are now able to produce in one hour what used to take half-a- day. In addition to the savings in time, there is less error- related waste of expensive parts/material. Production scheduling is also simplified. This automated system was relatively simple to implement and uses hardware currently available in the marketplace: the HP-1000, the Automation Unlimited Work Station, an HP-12792B 8-channel MUX, and the HPIB DVM.

1018. 'HIGH-LEVEL' PROGRAMMING USING THE HP MACRO/1000 ASSEMBLER

Dale S. Garcia, Senior Software Engineer
Technology Development Corporation
621 Six Flags Drive
Arlington, Tx. 76011

INTRODUCTION

In the earlier days of computers, before the development of modern 'structured' languages, a great deal of programming was performed using assembly language. As the computer industry matured, improved programming techniques were developed. As these techniques developed, additional languages were created to take advantage of these new ideas. In the 1970s, the concept of structured programming was introduced as a method of designing and developing software in a logical, systematic manner. Since then, a host of new languages have been developed such as C, Pascal, and Ada, each providing some new capability not found in existing languages. Some languages, such as FORTRAN and BASIC have evolved through various revisions to implement many of these new features. These new language capabilities greatly improve the speed and quality in which software is developed.

As high order languages developed, assembly language became used less except for where it was absolutely required. About the only major advance that assembly language has made is the introduction of the Macro-assembler. This new capability allows programmers to define macros to do a specific task. When a macro is invoked during the assembly, it is expanded, in a predefined manner, into several lines of code. The Macro-assembler has been under-utilized in its capability to greatly improve the efficiency of the software development cycle. With the proper development of a library of macros, tools are made available for use by the software developer. A library of macros has been developed for Structured Macro Assembly. This library is a collection of tools to implement all of the current structured programming constructs, along with several variations. It also has many extensions beyond the capability of higher order languages to handle assembly language-type functions. Most of the macros described in this paper are contained in this library.

This paper will discuss the macros used to implement these techniques along with additional macros to make programming and documentation easier. Due to time and space limitations, details of how the macros are written and the code they generate will not be discussed.

HP SUPPLIED MACROS

Hewlett-Packard supplies a library of macros with the Macro- assembler which perform many useful functions. The following macros are supplied in the library:

ENTRY	ADD	ROTATE
EXIT	SUBTRACT	ASHIFT
CALL	MAX	LSHIFT
	MIN	RESOLVE
IF		
ELSE	SETBIT	TEXT

ELSEIF	CLEARBIT	MESSAGE
ENDIF	TESTBIT	TYPE
	FIELD	STOP

Some of the HP supplied macros are briefly described in this paper. Additional documentation concerning HP supplied macros may be found in Appendix K of the MACRO/1000 Reference Manual.

SEQUENCE/INVOCATION MACROS

The sequence/invocation instructions usually perform a simple operation such as invoking a function/subroutine or performing an assignment operation. The sequence/invocation instructions are as follows:

DO	Invoke a subroutine without parameters
CALL	Invoke a subroutine with parameters
BEGIN	Begin subroutine w/o parms, retains entry point
SUBROUTINE	Begin subroutine with parms, retains entry point
FUNCTION	Begin function with parms, Same as SUBROUTINE
ENTRY	Provided by HP, Same as SUBROUTINE
RETURN	Return from subroutine or function
EXIT	Provided by HP, Same as RETURN
COPY	Copy word(s) from one location to another
INCR	Increment a variable
DECR	Decrement a variable
TRUE	Set variable(s) to logical .TRUE. value (-1)
FALSE	Set variable(s) to logical .FALSE. value (0)
CLEAR	Set variable(s) to a zero value
SET.BITS	Set selected bits as specified by mask word
CLEAR.BITS	Clear selected bits specified by mask word
CALC	Perform arithmetic/logical operations
FCALC	Floating point arithmetic/logical operations

The DO macro is used to invoke a subroutine that has no parameters, such as with the BEGIN macro. It is used to make the code more readable and generates a JSB instruction.

Example:

```
DO Read_request
```

The CALL macro generates the code to invoke a subroutine or function with up to ten parameters. The subroutine called should use the .ENTR sequence, such as generated by the SUBROUTINE, FUNCTION, and ENTRY macros.

Example:

```
CALL Get_status,Status_word,Return_code
```

Other forms of this macro (DCALL, LCALL, DLCALL) are also provided by HP to handle local and/or direct calls.

The BEGIN macro is used to enter a subroutine that has no parameters and does not need the return address resolved. It generates a NOP instruction to store the return address and retains the entry point name in order for the RETURN macro to generate the code to return from the subroutine. This macro is used with the DO and RETURN macros.

Example:

```
    BEGIN Read_request
```

The SUBROUTINE, FUNCTION, and ENTRY macros generate the .ENTR entry sequence of instructions for a subroutine or function with up to ten parameters. Each subroutine or function should have at least one corresponding RETURN macro to return to the calling routine.

Example:

```
Get_record   SUBROUTINE ^Buffer,^Record_length,^Error_code
Factorial    FUNCTION ^Number
Get_status   ENTRY ^Status_word,^Return_code
```

The RETURN macro is used to return from a subroutine or function that has been entered with the BEGIN, SUBROUTINE, FUNCTION, or ENTRY macro. The macro generates the return jump for the most recent subroutine/function macro. If a parameter is specified with the RETURN macro, the A-register will be loaded with the data as specified by the parameter before the return jump is made. A second parameter may also be specified to load the B-register. In this manner, a function may return a value to the calling routine.

Example:

```
    RETURN
    RETURN Value
    RETURN Status,Transmission_log
```

The COPY macro is used to copy a word from one location to another or will copy multiple words from one location to another. The first parameter specifies the source location. The second parameter specifies the destination location. The third parameter (optional) specifies the number of words to copy. If the length is not specified, the macro will generate a LDA and STA to copy one word. If the length is specified, the macro will use the MVW instruction, in which case, the source and destination parameters are location addresses.

Example:

```
    COPY Stop_flag,Program_flag
    COPY ^Data_buffer,^Name_field,Name_length
```

The INCR macro will generate the code to increment the value of a variable by a specified amount. The DECR macro will generate the code to decrement the

value of a variable by a specified amount. The value to increment or decrement is specified by the first parameter. The second parameter (optional) specifies how much the value is to be incremented or decremented. If not specified, the value will be incremented (for INCR) or decremented (for DECR) by one (1).

Example:

```
INCR Error_counter
DECR Words_left,Record_length
```

The TRUE macro sets all variables in the parameter list to a logical .TRUE. value (-1) which is compatible with FORTRAN LOGICAL*2. The FALSE macro sets all variables in the parameter list to a logical .FALSE. value (0).

Example:

```
TRUE Got_record,End_of_file,Process_data
FALSE Continue_proc,Read_data_rec
```

The CLEAR macro will clear all variables in the parameter list to zero.

Example:

```
CLEAR Record_length,Record_number,Record_index
```

The SET.BITS macro will generate the code required to set selected bits according to the bits set in a mask word. The bits set in the mask word will set the corresponding bits of the variable with the IOR instruction. All other bits will remain unchanged. The CLEAR.BITS macro will generate the code required to reset (clear) selected bits according to the bits set in a mask word. The bits set in the mask word will reset (clear) the corresponding bits of the variable.

Example:

```
SET.BITS System_flag,Error_bit
CLEAR.BITS System_flag,First_time_flag
```

The CALC macro generates the code to perform arithmetic and logical operations. The first parameter is the variable where the result of the operation is to be stored. The second parameter is the expression to be evaluated. Imbedded blanks are allowed in the expression if the string is in quotes. A constant number in the expression will be treated as a constant value and not an address. Eg: 3+4 will be treated as '=D3 + =D4'. An octal constant may be specified by appending a 'B' to the end of the number. For example, a '177400b' will generate the value of '=B177400'. The allowed operators for integer and single word operations are as follows:

OPERATOR	OPERATION
+	INTEGER ADDITION
-	INTEGER SUBTRACTION or UNARY NEGATE
*	INTEGER MULTIPLICATION

/	INTEGER DIVISION
&	LOGICAL AND
|	LOGICAL OR
\	LOGICAL EXCLUSIVE OR
~	LOGICAL NOT (COMPLEMENT)
<	LOGICAL SHIFT LEFT
>	LOGICAL SHIFT RIGHT
[	ROTATE LEFT
]	ROTATE RIGHT

Notes: '&' and '\' must be part of quoted string to properly work with the macro assembler.

The shift/rotate amount must be an integer value from 1 to 16. (not a variable or literal) eg: 6, not =D6 or Six to shift/rotate 6 bits.

Arithmetic, logical, and shift/rotate operators may be combined in an expression. A unary negate may be used for an operation by preceding the value to be negated by a minus (-) sign. Evaluation of the expression is from left to right. Parentheses are permitted for documentation purposes, but are ignored when evaluating the expression.

Example:

```
CALC Word_addr,'((-Byte_length-1) / 2) + Buffer_addr'
CALC Request_type,Request_code'&'~request_mask'\'177400B
CALC Device_stat,'((Interface_stat&377b)<8)|Card_stat'
```

The FCALC macro is similar to the CALC macro, but it generates single precision floating point instructions for arithmetic operations and double word instructions for the logical and shift/rotate operations. Similar macros may be written for extended precision operations. Mixed mode arithmetic is not supported.

ALTERNATION INSTRUCTIONS

The alternation instructions will conditionally execute a group of instructions depending on the state of a condition. There are two types of alternation instructions, simple and multiple. The simple type is in the form of IF..ELSE..ENDIF statements. The multiple alternation instruction is the CASE statement.

The IF macro is used to generate conditionally executed code. It is similar to the FORTRAN and Pascal IF Statement. The macro IF statement is used with the ELSE, ELSEIF, and ENDIF macros. The IF Statement will generate a JMP statement to a label used by these other macros. The following conditions may be tested:

Relationship	Operator
equal	= or EQ
not equal	<> or NE
greater than	> or GT
GT or equal	>= or GE
less than	< or LT
LT or equal	<= or LE

Example:

```
IF Return_code,EQ,System_failure
IF A,<=,Max_value
```

The ELSE macro is used in conjunction with the IF and ELSEIF statements. It will generate code to execute a block of statements when the logical condition for the corresponding IF statement is false. An ENDIF macro is used to terminate the ELSE block.

The ELSEIF macro will generate an ELSE statement followed by another IF statement without increasing the nesting level. Otherwise, the IF statement portion of the ELSEIF statement behaves like the IF statement. The ELSEIF statement must be followed eventually by an ELSE, ELSEIF, or ENDIF statement.

The ENDIF macro is used to terminate the block of statements for the preceding IF, ELSE, or ELSEIF. The ENDIF also reduces the nesting level by one.

Example:

```
IF A,EQ,Parity_error
  :                          ; code for IF goes here
ELSEIF A,EQ,Xmit_error
  :                          ; code for ELSEIF goes here
ELSE                         ; ELSE statement is optional
  :                          ; code for ELSE goes here
ENDIF
```

The IF statement is provided in the HP macro library. Several different IF statement variations are used to cover a broader range of conditions found when programming in assembly language. Additional IF macros are as follows:

```
IF.TRUE          Test if value is .TRUE. (MSB one)
ELSEIF.TRUE      ELSEIF form of IF.TRUE
IF.FALSE         Test if value is .FALSE. (MSB zero)
ELSEIF.FALSE     ELSEIF form of IF.FALSE

IF.ZERO          Test if value is equal to zero
ELSEIF.ZERO      ELSEIF form of IF.ZERO
IF.NOT.ZERO      Test if value is non-zero
ELSEIF.NOT.ZERO  ELSEIF form of IF.NOT.ZERO
```

IF.POSITIVE	Test if value is positive (MSB zero)
ELSEIF.POSITIVE	ELSEIF form of IF.POSITIVE
IF.NEGATIVE	Test if value is negative (MSB one)
ELSEIF.NEGATIVE	ELSEIF form of IF.NEGATIVE
IF.EVEN	Test if value is even (LSB zero)
ELSEIF.EVEN	ELSEIF form of IF.EVEN
IF.ODD	Test if value is odd (LSB one)
ELSEIF.ODD	ELSEIF form of IF.ODD
IF.BITS	Test selected bits specified by a mask If all corresponding bits set in the mask are also set in the variable, then the subsequent code will be executed.
ELSEIF.BITS	ELSEIF form of IF.BITS
IF.NOT.BITS	Test if selected bits are clear. If all corresponding bits set in a mask are clear in the variable, then the subsequent block of code will be executed.
ELSEIF.NOT.BITS	ELSEIF form of IF.NOT.BITS
IF.FLAG.SET	Test if I/O card flag is set for select code
ELSEIF.FLAG.SET	ELSEIF form of IF.FLAG.SET
IF.FLAG.CLEAR	Test if I/O flag is clear for select code
ELSEIF.FLAG.CLR	ELSEIF form of IF.FLAG.CLEAR

Example:

```
IF.FLAG.SET Phi
   DO Process_phi_intr
ELSIF.FLAG.SET Dma_complete
   IF.TRUE Read_request
      DO Cont_dma_read
   ELSE
      DO Cont_dma_write
   ENDIF
ELSEIF.FLAG.SET Dma_parity
   DO Device_clear
   GOTO $DMPR
ELSE
   DO Illegal_intr
ENDIF
```

The CASE routines implement the CASE structure. CASE statements may be nested up to 15 levels deep. The depth of the case statements does not affect the nesting level of the IF statements.

The DO.CASE.OF macro is used to initiate the case structure. The first parameter is a variable that will be used for the selection criteria for each case. An optional mask may be specified as the second parameter to mask out unwanted bits before each case is tested. A third parameter may be used to specify the length of the selector value.

The CASE.OF macro is used to specify each case that is being tested. If the variable (ANDed with the mask in the DO.CASE.OF macro) equals the value of the parameter specified in the CASE.OF statement, then the subsequent block of code is executed. If the values are not equal, then execution proceeds to the next CASE.OF, OTHERWISE, or END.DO.CASE statement for the case structure.

The OTHERWISE statement is used to specify the block of code to be executed if none of the previous case conditions were satisfied. This statement is optional.

The END.DO.CASE statement must be specified at the end of each case structure. It generates a label for the end of each case block to jump to when completed.

Example:

```
DO.CASE.OF Request_code,Req_code_mask
  CASE.OF Read_request
    DO Process_read
    :
  CASE.OF Write_request
    DO Process_write
    :
  CASE.OF Control_request
    DO Process_control
    :
  OTHERWISE
    INCR Error_counter
    DO Process_error
    :
END.DO.CASE
```

ITERATION INSTRUCTIONS

The iteration routines are a collection of routines to repetitively execute a block of code a number of times. The number of times a block of code is executed is dependent on the type of statement and the values of the parameters used. A block of code may be exited before the termination conditions are met by using a break statement.

The following macros are used to implement the iteration routines:

DO.UNTIL	Perform a block of code until a condition is met
END.DO.UNTIL	End of DO.UNTIL block. (post-test)
DO.WHILE	Execute code while condition is true. (pre-test)
END.DO.WHILE	End of DO.WHILE block.
DO.FOR	Execute code specified number of times (DO Loop)
DO.NEXT	End of DO.FOR. Do next iteration of loop.
DO.REPEAT	Repeat block of code a specified number of times
END.DO.REPEAT	End of DO.REPEAT block.

CYCLE.DO Skip rest of block, do next iteration of loop

BREAK.DO Terminate DO.UNTIL, DO.WHILE, DO.FOR, or DO.REPEAT

The DO.UNTIL macro will generate the instructions necessary to perform a block of code until a specified condition is met. The test for the condition is made at the end of the block, hence, a block is always executed at least once.

The END.DO.UNTIL macro terminates the DO.UNTIL block. It must be specified following the body of the DO.UNTIL block.

Example:

```
DO.UNTIL Buffer_index,>,Buffer_size
  :
END.DO.UNTIL
```

The DO.WHILE macro will generate the instructions to perform a block of code while a given condition is true. The test for the condition is performed at the beginning of the block.

The END.DO.WHILE macro terminate the the DO.WHILE block. It must be specified following the body of the DO.WHILE block.

Example:

```
DO.WHILE Process_flag,=,TRUE
  :
END.DO.WHILE
```

The DO.FOR macro generates the instructions to implement the FOR-NEXT loop. The first parameter is the variable to be used for storing the index value. The second parameter is the initial value to be assigned to the index variable. The third parameter is the termination value. The loop will terminate when the third parameter value is exceeded. The fourth parameter is the optional step size to increment or decrement the loop index value on each pass through the loop. Any of these values may be negative as long as the step size is consistent with the termination value. When the loop is terminated, the value of the loop variable will be incremented by the step size past the termination value. The number of passes through the loop will be calculated and maintained separate from the parameters and may not be subsequently modified.

The DO.NEXT macro generates the code to specify the next iteration of the previous DO.FOR. The loop variable is incremented by the step size and a check is then made to see if the loop has been executed the proper number of times. Alteration of any of the parameters specified in the DO.FOR after the loop has been entered will not effect the number of times that the loop will be executed.

Example:

```
DO.FOR Buffer_pointer,Begin_of_bufr,Bufr_length,Record_size
  :
DO.NEXT
```

The DO.REPEAT will generate the instructions necessary to repeat a block of code a specified number of times. Once the REPEAT block has been entered, changing the value of the parameter will not effect the number of times that the block will be executed.

The END.DO.REPEAT macro is used to terminate the DO.REPEAT block. It must be specified following the DO.REPEAT block.

Example:

```
DO.REPEAT Number_of_passes
  :
END.DO.REPEAT
```

The BREAK.DO macro is used to break out of an iteration block.

Example:

```
DO.UNTIL End_of_file,=,TRUE
  :
  IF Action_flag,=,Abort_loop
    BREAK.DO
  ENDIF
  :
END.DO
```

The CYCLE.DO macro is used to cycle through to the next iteration of a DO.WHILE, DO.UNTIL, DO.FOR, or DO.REPEAT loop.

Example:

```
DO.UNTIL End_of_file,=,TRUE
  :
  IF Action_flag,=,Skip_record
    CYCLE.DO
  ENDIF
  :
END.DO.UNTIL
```

DATA DECLARATION

The following macros are useful to allocate and initialize variables. Variable names may be declared using these macros to document the data type and to properly initialize the variable for the given data type. Since there is no automatic

data type conversion, operations on the data are the responsibility of the programmer. Each macro, except for CHARACTER, will generate one line of code for each variable in the variable list. The CHARACTER macro will generate three lines of code for each variable. These macros are designed to be compatible with their FORTRAN counterparts.

Macro	Generated Code	
INTEGER	DEC 0	
INTEGER.2	DEC 0	
INTEGER.4	DEC 0,0	
REAL	DEC 0.0	
REAL.4	DEC 0.0	
REAL.6	DEX 0.0	
REAL.8	DEY 0.0	
LOGICAL	OCT 0	
LOGICAL.2	OCT 0	
LOGICAL.4	OCT 0,0	
HOLLERITH	ASC w,xx...x	w = # words
CHARACTER	DEC b+100000B	b = # bytes
	DBL *+1	
	ASC b+1/2,xx...x	x = chars

Example:

```
INTEGER   I,J,K,Buffer_index,Buffer_word,Error_code
INTEGER.4 Two_words
REAL.4    Value,Old_value,Total_value
REAL.6    Extended_value
LOGICAL   Data_flag,End_of_file,Error_flag
```

The TABLE macro is used to create entries in a table. Several values may be contained in each table entry. The values may be any of the valid Macro/1000 language literal values or an indirect reference. An indirect reference is made by preceding a literal value or a variable name with an at (@) sign. This will produce the address of the value rather than the value itself.

Example:

```
ENTRY  TABLE =B12347,=D3,1.2,=S'a string',@=S'Ptr to string'
       TABLE @Prior,@Current,@Next
       TABLE @Command,@Mask,@Next_state,@Process_command,3
```

THE STATE MACHINE

A state machine may be created by using macros to define states and the conditions required to allow control to pass to another state. Before control is passed to the next state, an optional action routine may be executed. A parameter may also be passed to the action routine if required. The DEFINE.STATE macro is used to label and define the start of each state. The TRANS macro is used to define the

conditions to transfer to another state and to define the corresponding action routine and parameter. Several TRANS macros may be given for each state. The END.STATE.DEF macro is used to indicate the end of the defined state.

Syntax:

```
label  DEFINE.STATE
       TRANS val1,rel_op,val2,next_state,act_routine,act_parm
       :
       END.STATE.DEF
```

Where:

val1 and val2	are the values to compare
rel_op	is a relational operator (see IF macro)
next_state	is the state to transfer to if the given condition is true
act_routine	is the action routine to execute if the given condition is true
act_parm	is the parameter to pass to the action routine (if executed)

Example:

```
State0 DEFINE.STATE
       TRANS ,,,State1,Init_system,Terminal_no
       END.STATE.DEF

STATE1 DEFINE.STATE
       TRANS Temperature,<,Min_temp,State2,Low_temp,Temperature
       TRANS Temperature,>,Max_temp,State1,Cool,Max_temp
       TRANS Fluid_level,<,Min_level,State1,Add_fluid
       TRANS Fluid_level,>,Max_level,State1,Stop_fluid
       TRANS Pressure,>,Max_pressure,State3,STOP_SYSTEM
       TRANS ,,,State1,Wait,30
       END.STATE.DEF

STATE2 DEFINE.STATE
       TRANS Temperature,>=,Min_temp,State1
       TRANS Pressure,>,Max_pressure,State3,STOP_SYSTEM
       TRANS ,,,State2,Wait,5
       END.STATE.DEF

STATE3 DEFINE.STATE
       TRANS Resume_flag,=,Resume,State0
       TRANS ,,,State3,Wait,1
       END.STATE.DEF
```

SYNTAX DIRECTED TEXT PARSER

A syntax directed text parser, similar to TPARSE found on the VAX-11/780 (Digital Equipment Corp.), may be created using macros. The syntax directed text parser is useful for the development of command interpreters, compilers, and translators. It is also useful for interactive programs to provide a greater flexibility for user input. The text parser operates like the state machine except that transitions are made depending on the input text string. Before control is passed to the next state, an optional action routine may be executed. The symbol matched will be passed to the action routine. A parameter may also be passed to the action routine, if required. The STATE macro is used to label and define the start of each state. The TRAN macro is used to define the condition required to transfer to another state and to define the corresponding action routine and parameter. Several TRAN macros may be given for each state. The END.STATE macro is used to indicate the end of the definition for the state. The following symbol types may be checked in the string:

Symbol Type	Character(s) Matched
'$x'	Any particular character
$$ANY_CHAR	Any single character
$$ANY_ALPHA	Any alphabetic character
$$ANY_DIGIT	Any numeric character
$$ANY_STRING	Any alphanumeric string
$$ANY_SYMBOL	Any valid symbol name
$$ANY_BLANK	Any string of blanks
$$ANY_DECIMAL	Any decimal number
$$ANY_OCTAL	Any octal number
$$ANY_HEX	Any hexadecimal number
'keyword'	Any particular keyword string
$$NIL_STRING	Match anything (string pointer not moved)
$$END_STRING	End of input string

Syntax:

```
label  STATE
       TRAN symbol_type,next_state,act_routine,act_parm
       :
       END.STATE
```

Where:

symbol_type	is one of the valid symbol types.
next_state	is the state to transfer to if the given symbol type is matched. $EXIT is used to exit the parser. $ERROR is used for an error exit from the parser.
act_routine	is the action routine to execute if the given symbol type is matched.
act_parm	is the parameter to pass to the action routine (if executed).

Example:

```
State0 DEFINE.STATE
       TRAN 'MARY',State1,Proc_name,1
       TRAN 'JACK',State1,Proc_name,2
       TRAN 'JILL',State1,Proc_name,3
       TRAN $$NIL_STRING,$ERROR,Proc_error,1
       END.STATE

State1 DEFINE.STATE
       TRAN 'HAD',State2,Proc_verb,1
       TRAN 'FETCHED',State2,Proc_verb,2
       TRAN 'AND',State0
       TRAN $$NIL_STRING,$ERROR,Proc_error,2
       END.STATE

State2 DEFINE.STATE
       TRAN 'A',State2
       TRAN 'LITTLE',State2,Proc_descr,1
       TRAN 'BIG',State2,Proc_descr,2
       TRAN $$NIL_STRING,State3
       END.STATE

State3 DEFINE.STATE
       TRAN 'LAMB',State4,Proc_object,1
       TRAN 'PAIL',State4,Proc_object,2
       TRAN $$NIL_STRING,$ERROR,Proc_error,3
       END.STATE

State4 DEFINE.STATE
       TRAN $$END_STRING,$EXIT,End_of_story
       TRAN $$NIL_STRING,$ERROR,Proc_error,4
       END.STATE
```

CONCLUSION

With the use of macros, programming in assembly language will become both more productive and easier to maintain. The concepts and examples given should provide an insight as to what may be done using the power of the Macro-assembler. This paper is intended as an overview of what can be done and is by no means complete or totally descriptive. Additional macros may be needed, in some cases, to support the macros mentioned in this paper and to declare & initialize required assembly time variables.

1019. TAKING ADVANTAGE OF FORMS MODE FOR DATABASE ENTRY APPLICATIONS

R. Miller and J. Craft
Martin Marietta Aerospace
Michoud Div Dept 3437
P.O. Box 29304
New Orleans LA 70189

THE NEED FOR AN EASY TO USE DATA ENTRY SYSTEM:

Most of the database applications we are involved in are used to track project statuses or man-power requirements. Updates to these databases are performed via interactive terminal input. When constructing the terminal data entry routines, we were faced with the following problems;

1: Our group's primary task is process control software, not database applications. Because of this, we would not have a lot of time to train users or to assist with data entry problems.

2: Our average user is a secretarial type with minimum computer orientation.

3: Memory in our system is configured for our primary task, process control software development, and could not be re-configured. Maximum size of the database application programs was 32K.

We needed a data entry routine which would meet the following criteria;

1: It must be easy to train users.
2: It must be as easy to use as possible.
3: It must not greatly affect program size.
4: It must be flexible enough to handle all types of data.
5: It must be able to handle data entry modifications without a lot of re-programming.

We had a lot of problems meeting more than two of our criteria at a time until we got ahold of two routines which constructed and displayed forms on a 264x terminal. Without any modifications they met all of the criteria, but they also added a whole new set of problems. The major ones were;

1: The forms construction program was very un-forgiving. Any mistake could totally destroy the desired output.
2: It was impossible to determine if a form was accidently erased while in use by an application program. This made it extremely difficult to exit the program.
3: Output of error messages to a terminal in forms mode involved quite a bit of complicated code.

However, these routines provided us with an excellent base for our data entry routines. The following routines are the current results of our efforts to develop a data entry system to meet our criteria.

BASIC PARTS:

There are three basic parts to our forms package;
First; FORMS, A program to construct and edit the forms, Second; FFORM, A subroutine which controls forms IO, and Third; CFORM, A subroutine which handles non-forms IO to a CRT in forms mode.

FORMS PROGRAM - Creating and Editing:

Form Files:

We'll start at the logical place, forms construction. Forms are created as File Manager type 30 files by the program FORMS. Type 30 was selected just to give us a non-standard file type to eliminate problems like trying to run the editor on a form, or passing a non-form file to our form display routine.

To create or edit a form, the user runs FORMS, selects the Modify option, and then enters a file NAMR. If the file can not be found, the create form mode is assumed and a file is created. If the file is found, it is opened and the first record is read and re-written to verify read/write access to the file. This eliminates a problem we had of spending 30 minutes editing a form only to discover we couldn't re-write the result back to the original file.

Softkeys:

Once the program has verified capability to access the form file, the CRT softkeys are programmed to enable the user to quickly define the unprotected fields on the screen. The display shown in figure 1 is output to the CRT. Unfortunately, this is the only time the user can view this display. Once forms editing begins, there is no way to output any help messages, so you must remember what each softkey does.

When RETURN is pressed the screen is cleared and, if in modify mode, the current version of the form is output to the CRT. The screen may then be changed as desired with the Edit group, Display group, or Alpha-numeric keypad. When the desired form is completed, the user presses the ENTER key to read the form into the file.

Edit Rules and Practices:

There are some rules:

1: To abort an edit without file re-write, put an XX in the top left corner of the screen before pressing ENTER. This provides an escape route for a serious error such as pressing the clear display key instead of the delete character key.

2: To delete an unprotected field, start with the character before the field and use the delete character key. Starting with the first position of the field, or using the space bar will not delete the undisplayed field control characters.

3: There must be less than 20 consecutive blank lines in any given form, as the program stops the screen to file copy when it encounters this condition.

4: If the screen is to be cleared before the form is output by an applications program, put a C in the top left corner of the screen. Otherwise the screen will be cleared only from the first non-blank line of the form. This feature is extremely useful when the top lines of the form are identical as in figures 2 and 3. The top section of both screens is in one file while the bottom sections are in two different files. To produce the screens shown, the top section is displayed first, then one of the bottom sections is displayed. Although this method of combining files to produce one screen does conserve some disc space, the primary reason this is done is to keep forms editing at a minimum if it is ever necessary to change the top section of the screen.

A good practice to follow is to leave the top line blank so that the application programs can use it to identify the form or a process step. Quite often in our application the same form is used in different tasks. Before we installed this feature, our users would sometimes lose track of which task was being performed.

It's also a good idea to leave a couple of blank lines at the bottom of the screen for application program messages. Message output techniques will be discussed later.

FORMS PROGRAM - Forms Check-Out:

After the form is created, the most obvious thing to do is to verify that it has been created correctly. Originally this involved dumping the form to the terminal and insuring that the form was displayed correctly and that the fields were of the proper alpha/numeric type. This method left the terminal in forms mode and a hard reset had to be done to return it to normal operations. To make check-out easier, the TRY routine was added to the program FORMS.

The "TRY" routine actually simulates a program, which displays a single form or as many as 10 forms to a CRT. This is done to check the form's display and to verify the unprotected fields for proper length and data types. This routine is extremely useful in preparing the forms for a program. If any of the forms are incorrect the programmer can return to the MODFIY routine, make the necessary corrections, and TRY the forms again. It is also useful during the coding stage of a program, because the programmer can TRY the forms at any time to get the correct field size and type or location of data in the forms.

Prompts for the "TRY" routine are;

```
Entry expected is:
      Form-File-Namr <,flag ,ntab>
        flag = 1 to advance to next form without pressing ENTER
        ntab = Number of tabs to output

Enter file name # 1:

The input should consist of all the forms to be verified
```

and a blank entry stops the input and starts the display.

Definition of input:

Form-File-Namr is the file names of the forms to be displayed.

flag controls how to advance through the forms.
If flag is set to zero (0) than try this form before advancing to the next form by pressing ENTER.
This is useful in stepping through a multiple set of forms.
If flag is set to one (1) than the next form is displayed without pressing ENTER.

NOTE: When displaying a single form or the last form of multiple set of forms, the flag should be set to zero (0) to try the form or forms.

ntab If field positioning is desired, ntab is the field number minus one.

NOTE: If flag is not zero than ntab is ignored.

Figure 4 shows some input examples for the TRY routine.

FORMS PROGRAM - Listing Forms:

The LIST routine in the FORMS program grew out of a necessity to produce hard copies of the forms for documentation and user approval purposes. Attempts to dump a form file to the 2608A printer resulted in unreadable text as the inverse video and unprotected field control codes were printed. The LIST routine strips these characters from the output and replaces the spaces in the unprotected fields with a vertical line as an aid in determining the field locations.

Prompts for the LIST routine are;

Enter List LUN/File namr:

User may enter a logical unit number or a file namr as the list output device.

NOTE: If a file namr is entered and the file already exists the program will request permission to overwrite the file.

After the list device is selected, the program will request form file namrs to be listed. A blank entry terminates the list routine.

Figure 5 shows a copy of a print listing.

APPLICATION SUBROUTINES = FFORM:

The subroutine FFORM is used to display forms, read forms, and reset the terminal to non-forms mode. The calling sequence is;

```
CALL FFORM (NAME, NCRN, NBUF, LEN, MODE, NCRT)
```

NAME and NCRN are used for display form mode to define the form file name and cartridge reference number.

NBUF is the data buffer for form read functions. This buffer must be sized large enough to contain all the data from the screen including the field separators. However, the field separators are stripped from the buffer before the data is returned.

LEN is used for display and read forms mode. When the call is made, it is the length of the data buffer. The return value depends upon the function selected as described below;
Display: Zero if no error, otherwise FMP negative error code.
Read: The word length used in the data buffer is returned.
A return of zero (0) indicates that no data was read from the screen. Usually this means that the screen was inadvertently cleared by the user. A return of minus one (-1) indicates that the screen was successfully read, however all the unprotected fields were blank. We typically treat this condition as a terminal time-out condition.

MODE defines the function or functions to be performed.
A layout of this parameter is shown in figure 6.
Bit 2 on selects the display form function,
Bit 1 on selects the read form function, and
Bit 0 on selects the reset terminal function.
Since functions are selected by certain bits, the user may issue a call which performs multiple functions such as display and read a form. In this case, the return value of LEN would be determined by the first un-successful, or the last successful function performed.

Depending upon the function selected, the MODE parameter may also be used to invoke the following options;

1: In the display function up to 78 characters of identifying text may be output on the top line of the terminal. To do this, set bit 3 of MODE on, put the text in the data buffer NBUF, and set LEN to the word length of the text. When this option is used, it is a good idea not to select the read form function as the text buffer will be used to return the data from the form and is normally too small for input operations.

2: In the read function the cursor may be positioned to the beginning of any of the first 128 fields on the screen by setting bits 14 thru 8 of MODE to the field number

minus one.

3: Also in the read function the terminal is normally set to time-out in five (5) minutes. This is done to prevent the terminal from being tied up if a user leaves in the middle of a session. To set infinite time-out set bit 15 of mode on. In either case, infinite or five-minute time-out, after the read form operation is complete the original time-out value for the terminal is restored.

NCRT defines the terminal logical unit number for the IO operations.

Figures 7 thru 10 show various calls to this routine and the resultant screen display.

APPLICATION SUBROUTINES - CFORM:

The subroutine CFORM is used to control non-form IO on a terminal which is in forms mode.

The calling sequence is;

CALL CFORM (NOPT, NCRT, NPOS)

NOPT is the function to perform where;
- 1 = Turn forms mode off and line position the cursor.
- 2 = Turn forms mode on and field position the cursor.
- 3 = Clear screen from a specified line number, then field position the cursor.
- 4 = Field position the cursor then output spaces to form.

NCRT is the terminal logical unit number

NPOS defines the positioning information for the function selected. The left byte contains the field number minus one for all functions which perform field positioning. This value cannot exceed 79. The right byte contains the line position for functions 1 and 3. This value cannot exceed 23.

For function 1, the sign bit is used to enable or disable the keyboard while the forms are off. The keyboard will be disabled unless this bit is on. We typically leave the keyboard disabled to output informational or error messages and only enable it if we are going to request a single non-form user input such as;

IS THIS THE CORRECT RECORD? (Yes or No):

For function 4, the right byte is the number of spaces to output. If this value is set to 255, then the unprotected fields of the form are cleared from the cursor position.

A typical use of functions 1 and 2 is shown in figure 11. Here the

forms mode is turned off in order to output an error message and then forms mode is turned back on.

The typical use for function 3 is to clear the error messages. And the typical use for function 4 is to clear a constantly displayed form.

FORMS TECHNIQUES:

User Training:

Usually a ten to twenty minute session with a user is all the training that is required. Also, because it is so easy, we normally only have to train one user from a group and then that user trains the others in his group. Our typical user has no idea of what program is running or what database is being accessed. He merely sits down at an available terminal and logs on. From that point on all his decision making is performed thru forms entry. He has only three things to remember;

1: His user ID and password,
2: Press ENTER when the form is filled in, and
3: Press RETURN and re-type the last field if the cursor won't move.

The last item is due to the fact that forms are using the extended edit capabilities of the terminal and if the alpha/numeric type of the character entered by the user does not meet the field type, the terminal beeps and keeps the cursor in the position where the character is entered. There is a caution on this. The HP 238x terminals do not have extended edit capabilities. If you are using those terminals, you'll have to perform alpha/numeric character validation within the application program.

Dealing With Form Changes:

Unless you're living in a totally different world than I am, one of the life's constants is user requested changes. This used to cause all kinds of minor headaches in re-formatting prompt messages and input statements. Thru the use of forms, we've been able to greatly reduce the problem.

Figure 12 shows an original version of a user entry form. Note the position of the fields FACTOR and SCHEDULE TYPE. Figure 13 shows a modification to this form to meet additional user requirements. Here a function escape method has been provided, two new fields dealing with installation data have been added, and the SCHEDULE TYPE and FACTOR fields have been repositioned to correspond with the user input document.

Figure 14 is the code used to read both forms. As can be seen, the only changes required for the modified form were a DECODE and a FORMAT statement. Granted, program changes had to be made to handle the additional three fields, but what we're concerned with here is only those changes required to handle input changes. Our average time to make this type of change is around 15 minutes. Most of that time is spent editing the input form.

Outputting Data To Forms:

Data may be output to a form using standard FORTRAN formatted write statements. If necessary, the cursor may be positioned to the start of a field using function 4 of CFORM. Special consideration must be given when outputting more than 132 characters to a form since this exceeds the normal buffer limits of the terminals. One way around this is to use the HP subroutine LGBUF to increase the IO buffer size. Another method is to output less than 132 characters with the last character being an underscore (_), and then output the remaining characters.

BENEFITS AND SUMMARY:

There are two major benefits derived thru our use of forms;

First: User appreciation and acceptance of our system has greatly increased. A major factor in this increase is the minimal amount of entry rules that have to be learned. We spend less time training the users and assisting them with problems in running their applications. We have received several comments on how easy our system is to use compared to other systems they have used.

Second: The programmers in our group have more time to work on the 'guts' of the program. Major reasons for this are the reduced time spent with the user, and the fact that CRT IO has become a matter of displaying the correct form and reading or writing the data.

Use of forms relieves the programmer of a lot of the burden in formatting terminal IO. Also, if the terminal has extended edit capabilities, the necessity for verifying the data type inside the application program is eliminated. Use of forms need not be restricted to data entry either. A form can be output to the terminal and then filled in with data. Here again, the programmer is relieved from writing FORMAT statements to get the desired results.

Although we developed it for our database applications, forms may be used in any program which performs a lot of terminal IO. It is a method which is easy to program, easy to train, and easy to use.

```
f1 - Start unprotected field with '['

f2 -  Stop unprotected field with ']'

f3 - Start unprotected field

f4 -  Stop unprotected field

f5 - Start unprotected field, numerics only

f6 - Start unprotected field, alpha only

f7 - Start unprotected field with '[', numerics only

f8 - Start unprotected field with '[', alpha only

Press RETURN when ready to continue: _
```

Figure 1: Softkey settings for form creation/edit.

```
Enter Report Function - Tool ID is OK

  Exit[ ]   Add[ ]   Change[ ]   Copy[ ]   Display[ ]   Report[X]   Delete[ ]
 ------------------------------------------------------------------------------
    TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
              [T27K1234     ]        [NEW ]      [  1]        [  2]
 ==============================================================================
To abort this function put a A in this box ---------------> [ ]

               [_] Deleted tool list
               [ ] Active  tool list
               [ ] Display Tool ID above on this CRT
               [ ] Print using Tool ID above
               [ ] Print changes since this date      [  /  /  ]

    Press  ENTER  to read form.
```

Figure 2: Example # 1 of form output with identification line.

```
Enter Report Function - Tool ID is Invalid

  Exit[ ]   Add[ ]   Change[ ]   Copy[ ]   Display[ ]   Report[X]   Delete[ ]
 ------------------------------------------------------------------------------
    TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
               [T27K        ]        [    ]     [   ]       [   ]
 ==============================================================================
To abort this function put a A in this box ----------------> [ ]

              [_]  Deleted tool list
              [ ]  Active  tool list
              [ ]  Display Tool ID above on this CRT
              [ ]  Print using Tool ID above
              [ ]  Print changes since this date      [  /  /  ]

     Press  ENTER  to read form.
```

Figure 3: Example # 2 of form output with identification line.

Example # 1: Display a single form.

```
Enter file name # 1: FORM1
Enter file name # 2: [CR]
```

Example # 2: Display a single form and field position the cursor.

```
Enter file name # 1: FORM1,,9
Enter file name # 2: [CR]
```

Example # 3: Display a three forms one at a time

```
Enter file name # 1: FORM1
Enter file name # 2: FORM2
Enter file name # 3: FORM3
Enter file name # 4: [CR]
```

Example # 4: Display a three forms as one screen
And field position cursor.

```
Enter file name # 1: FORM1,1
Enter file name # 2: FORM2,1
Enter file name # 3: FORM3,,9
Enter file name # 4: [CR]
```

Figure 4: TRY routine input examples.

```
****** List of Form FNAM1 ******

Enter Person's NAME
      LAST                        FIRST                INT
|||||||||||||||||||     |||||||||||||||||||||||     ||

******
****** List of Form FNAM2 ******

ADDRESS ||||||||||||||||||||||||

     CITY               STATE        ZIPCODE
|||||||||||||||   |||||||||||||||   ||||||

******

NOTES:
    ****** List of Form namr ******   Indicates start of form
    ******       Indicates end   of form
    |            Indicates unprotected field position
```

Figure 5: LIST routine output example.

Keyboard Control	Field Positioning	Not Used	H	D	R	X
15	14 8	7 4	3	2	1	0

Bit	Usage
X	Reset Terminal to normal mode
R	Read form
D	Display form
H	Output text header line (Display form mode only)

Keyboard Control and Field Positioning use for Read form mode only

Figure 6: MODE parameter for FFORM

```
      LEN=10
      MODE=4
      CALL FFORM (NAME1,0,TEXT1,LEN,MODE,NCRT)
      IF (LEN .NE. 0) GO TO error
```

```
  Exit[_]   Add[ ]   Change[ ]   Copy[ ]   Display[ ]   Report[ ]   Delete[ ]
  ------------------------------------------------------------------------
     TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
                [            ]        [    ]      [   ]       [   ]
  ========================================================================
  To abort this function put a A in this box ------------------------> [ ]
            Schedule Type  [ ]   (S=Standard, C=Compressed, X=Expanded)
Number of Phase Indentures [  ]
                    Factor [     ]
      Gross Standard Hours [        ] (Per Unit)        Installation
                Start Date [  /  /  ]                 #Phases   Start Date
              Complete Date [  /  /  ]                  [ ]     [  /  /  ]

Press  ENTER  To Read Form
```

Figure 7: FFORM Display form option without text ID line.

```
      DATA ITEXT / 'Enter Tool Data      ' /
       .
      LEN=10
      MODE=12
      CALL FFORM (NAME1,0,TEXT1,LEN,MODE,NCRT)
      IF (LEN .NE. 0) GO TO error

Enter Tool Data

  Exit[_]   Add[ ]   Change[ ]   Copy[ ]   Display[ ]   Report[ ]   Delete[ ]
  ------------------------------------------------------------------------------
     TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
                [            ]        [    ]      [   ]       [   ]
  ==============================================================================
  To abort this function put a A in this box --------------------------> [ ]
            Schedule Type  [ ]   (S=Standard, C=Compressed, X=Expanded)
Number of Phase Indentures [  ]
                    Factor [     ]
      Gross Standard Hours [        ] (Per Unit)         Installiation
                Start Date [  /  /  ]                  #Phases   Start Date
              Complete Date [  /  /  ]                   [ ]     [  /  /  ]

Press  ENTER  To Read Form
```

Figure 8: FFORM Display form option with text ID line.

```
      LEN=400
      MODE=2
      CALL FFORM (0,0,NBUF,LEN,MODE,NCRT)
      IF (LEN .EQ.  0) GO TO error
      IF (LEN .EQ. -1) GO TO no input

Enter Tool Data

  Exit[_]   Add[ ]   Change[ ]   Copy[ ]   Display[ ]   Report[ ]   Delete[ ]
  ------------------------------------------------------------------------------
     TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
                [            ]        [    ]      [   ]       [   ]
  ==============================================================================
  To abort this function put a A in this box --------------------------> [ ]
            Schedule Type  [ ]   (S=Standard, C=Compressed, X=Expanded)
Number of Phase Indentures [  ]
                    Factor [     ]
      Gross Standard Hours [        ] (Per Unit)         Installiation
                Start Date [  /  /  ]                  #Phases   Start Date
              Complete Date [  /  /  ]                   [ ]     [  /  /  ]

Press  ENTER  To Read Form
```

Figure 9: FFORM Read form option with no field positioning.

```
      LEN=400
      MODE=12*256+2
      CALL FFORM (0,0,NBUF,LEN,MODE,NCRT)
      IF (LEN .EQ.  0) GO TO error
      IF (LEN .EQ. -1) GO TO no input

Enter Tool Data

  Exit[ ]   Add[ ]   Change[ ]   Copy[ ]   Display[ ]   Report[ ]   Delete[ ]
  ------------------------------------------------------------------------
     TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
                [           ]        [    ]     [   ]       [   ]
  ========================================================================
  To abort this function put a A in this box --------------------------> [ ]
           Schedule Type  [_]   (S=Standard, C=Compressed, X=Expanded)
Number of Phase Indentures [  ]
                    Factor [     ]
      Gross Standard Hours [        ] (Per Unit)        Installiation
                Start Date [  /  /  ]                 #Phases   Start Date
              Complete Date [  /  /  ]                  [ ]     [  /  /  ]

Press  ENTER  To Read Form
```

Figure 10: FFORM Read form option with field positioning.

```
      CALL CFORM (1,NCRT,18)
      WRITE (NCRT,1) IERR,IESC,IESC
      CALL CFORM (2,NCRT,0)
    1 FORMAT ('DBGET Error on Dataset TFRPD1 = ',I6,//,
     >        'Please Record Error Message.  ',
     >        'Then press ',A1,'&dB ENTER ',A1,'&d@')

Enter Tool Data

  Exit[_]   Add[X]   Change[ ]   Copy[ ]   Display[ ]   Report[ ]   Delete[ ]
  ------------------------------------------------------------------------
     TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
                [T27K1234    ]       [NEW ]     [  1]       [  2]
  ========================================================================
  To abort this function put a A in this box --------------------------> [ ]
           Schedule Type  [S]   (S=Standard, C=Compressed, X=Expanded)
Number of Phase Indentures [2 ]
                    Factor [.7   ]
      Gross Standard Hours [289     ] (Per Unit)        Installiation
                Start Date [6 /25/85]                 #Phases   Start Date
              Complete Date [11/15/85]                  [ ]     [  /  /  ]

Press  ENTER  To Read Form

        DBGET Error on Dataset TFRPD1 = 114
        Please Record Error Message.  Then press  ENTER
```

Figure 11: CFORM Turn off forms to display an error message.

```
Enter Tool Data

  Exit[ ]   Add[X]   Change[ ]   Copy[ ]   Display[ ]   Report[ ]   Delete[ ]
  ------------------------------------------------------------------------
     TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
                [T27k1234    ]        [NEW ]      [  1]       [  2]
  ========================================================================

Number of Phase Indentures [_]           Factor [     ]
           Schedule Type  [ ]   (S=Standard, C=Compressed, X=Expanded)
      Gross Standard Hours [       ] (Per Unit)
                Start Date [  /  /  ]
             Complete Date [  /  /  ]

Press  ENTER  To Read Form
```

Figure 12: Original Tool Data Entry Form.

```
Enter Tool Data

  Exit[ ]   Add[X]   Change[ ]   Copy[ ]   Display[ ]   Report[ ]   Delete[ ]
  ------------------------------------------------------------------------
     TOOL ID:   Tool Number          Revision   From-Unit    To-Unit
                [T27K1234    ]        [NEW ]      [  1]       [  2]
  ========================================================================
  To abort this function put a A in this box ------------------------> [ ]
           Schedule Type  [_]   (S=Standard, C=Compressed, X=Expanded)
Number of Phase Indentures [  ]
                   Factor [     ]
      Gross Standard Hours [       ] (Per Unit)        Installiation
                Start Date [  /  /  ]                #Phases   Start Date
             Complete Date [  /  /  ]                  [ ]     [  /  /  ]

Press  ENTER  To Read Form
```

Figure 13: Re-structured Tool Data Entry Form.

```
      DIMENSION NBUF(400),ISDATE(3),ICDATE(3)
1     FORMAT (29X,I1,F5.0,A1,F8.0,6(I2))
      .
      .
      LEN=400
      MODE=11*256*2
      CALL FFORM (0,0,NBUF,LEN,MODE,NCRT)
      CALL CFORM (3,NCRT,16)              ! Clear previous messages
      IF (LEN .EQ.  0) GO TO 8000
      IF (LEN .EQ. -1) GO TO 4000
      DECODE (56,1,NBUF) NPHASE,FACTOR,ITYPE,GROSS,ISDATE,ICDATE



      DIMENSION NBUF(400),ISDATE(3),ICDATE(3),IIDATE(3)
1     FORMAT (29X,2A1,I2,F5.0,F8.0,6(I2),I1,3(I2))
      .
      .
      LEN=400
      MODE=11*256*2
      CALL FFORM (0,0,NBUF,LEN,MODE,NCRT)
      CALL CFORM (3,NCRT,16)              ! Clear previous messages
      IF (LEN .EQ.  0) GO TO 8000
      IF (LEN .EQ. -1) GO TO 4000
      DECODE (65,1,NBUF) IABT,ITYPE,NPHASE,FACTOR,GROSS,
     >                   ISDATE,ICDATE,IPHASE,IIDATE
```

Figure 14: Form read code for figures 12 and 13.

1020. INCOMING MATERIAL MANAGEMENT AND VENDOR QUALITY CONTROL: AN HP1000 CASE STUDY

G.R.Nevogt
G.L.McCrory
ITT Aerospace/Optical Div.
3700 E. Pontiac St.
Ft. Wayne IN 46801

In December of 1983 a government contract was awarded to ITT's Aerospace/Optical Division for the production of the Single Channel Ground and Airborne Radio System (SINCGARS). A new production facility was dedicated for this effort, and in May of 1984 the new facility was staffed in Fort Wayne, Indiana. It was Quality Engineering's desire to implement a computer which would automate many of the tasks traditionally done manually. For planning purposes, it was assumed that the system should be capable of processing seventy (70) incoming lots of material per day and that the material would be routed to one of eight inspection stations upon receipt. This paper addresses the tasks which are performed by this system in the Purchased Material Inspection (PMI) area at ITT's SINCGARS production facility. Among these tasks are:

(1) The creation and maintenance of inspection instructions for all purchased material used in the radio,

(2) The maintenance of information on each inspection lot in the department including current location, status, and priority level,

(3) The storage of statistics on parametric test data which could be used at a later date for the plotting of histograms to be used for trend analysis and process capability studies,

(4) The accumulation of inspection histories by part number and vendor, contributing to the determination of sampling requirements per MIL-STD-105D.

CHOOSING THE SYSTEM

Several factors were deemed essential in the determination of which computer system would be chosen for ITT A/OD's SINCGARS facility:

(1) A proven "track record" had to exist, with respect to service representative response to customer inquiries and problems,

(2) The system chosen should possess a history of performing in a similar applications' environment reliably. Downtime was, of course, a major concern as production capability is directly linked to the availability of parts,

(3) Relatively inexpensive applications software and software consulting should be available from a vendor located in close proximity to the facility,

(4) The system should be user friendly, as the population of users would be non-computer oriented.

It was decided after extensive research that an HP1000 was the computer which best satisfied our requirements. Our attention then turned to locating an applications software vendor, familiar with the Quality requirements involved in a program governed by MIL-Q-9858. Automated Technology Associates,Inc.(ATA), founded by a nucleus of Quality Management and laboratory automation personnel with expertise in Hewlett-Packard computer applications, was chosen to supply the software which would allow Purchased Material Inspection to efficiently carry out the required tasks.

INSPECTION INSTRUCTION MANAGEMENT

MIL-Q-9858 requires thorough procedures and instructions for all operations from procurement of material to delivery of the end product. Thus, the problems faced by preparing to inspect more than 1500 different piece parts in 12 months would seem a formidable task. Our objective was to be able to create and efficiently maintain approximately 1500 inspection instructions while at the same time minimizing disk storage requirements.

In discussing the problems of creating the work instructions with ATA, a program referred to as Technical Instruction Manager (T.I.M.) was proposed as a solution to the task.

Automation of this task was based on the concept that inspection instructions for like material would be very similar with only minor variations from part to part. Some examples;

- Resistance tests on carbon composition resistors are exactly the same except for value and tolerance specifications.

- Mechanical measurements made on screws are exactly the same except for thread and size specifications.

This concept allows boilerplates to be created for general types of inspection. Variable files containing part specific information would be merged into the boilerplates creating unique inspection instructions. See figures A and B.

Boilerplates and variables files can be organized using the CI hierarchical file system to facilitate efficient management. The main benefit of using the hierarchical file system was based on being able to efficiently maintain the files that, when merged, represent several thousand pages of work instructions. Boilerplates are created using EDIT, for each type of test to be performed in the inspection department. Variable files are also created using EDIT, for each part number to be inspected. Variable files control what boilerplates are used and sets the variable values to be inserted. Variable files representing different types of material are grouped into sub-directories by commodity. Organizing the files in this manner facilitates creation of similar instructions very efficiently. For example: The variable file for a specific part may be copied to create another variable file using the CO command. The minor changes required in the new variable file, as defined by the material specification, are readily

```
..** /WI/HEADER.BP
..** <<841112.1507>>
                  ***********************************
                  ***          ITT A/DD           ***
                  ***    PMI WORK INSTRUCTION     ***
*****************************************************************
!PART NUMBER: <<PN          >> SOURCE DOC. REV: <<REV          >>!
!-------------------------------------------------------------------!
!W/I REV: <<WIREV            >> DEVICE TYPE: <<TYPE             >>!
!-------------------------------------------------------------------!
!PLANNED BY: <<NAME          >> APPROVAL:                          !
!-------------------------------------------------------------------!
!EQUIPMENT REQUIRED: <<EQ1                  >>,<<EQ5               >>!
!                    <<EQ2                  >>,<<EQ6               >>!
!                    <<EQ3                  >>,<<EQ7               >>!
!                    <<EQ4                  >>,<<EQ8               >>!
!-------------------------------------------------------------------!
!SPECIAL NOTES: <<NOTE1                                             >>!
!               <<NOTE2                                             >>!
!               <<NOTE3                                             >>!
!               <<NOTE4                                             >>!
*****************************************************************
<<MESSAGE                                                          >>

..1=CALIBRATION DATE VERIFICATION

      A) Check all equipment calibration stickers and verify that
         calibration due dates are not exceeded.If the calibration
         due date is exceeded, DO NOT PROCEED with testing. Notify
         supervisor,technician,or engineer immediately!!
```

Figure A

```
..** TIM SWITCH VARIABLE FILE.
..** CREATED BY G.L. McCrory <<850522.1219>>
..** /WI/SWITCH/A3011234.PN
..*
..** SET VARIABLES FOR THE HEADER
..*
..SY=PN=A3011234-1
..SY=REV=G
..SY=WIREV=28 JANUARY 1989
..SY=TYPE=RELAY
..SY=NAME=J.C. SHUTT
..SY=NOTE1=
..SY=NOTE2=
..SY=NOTE3=
..SY=NOTE4=
..SY=EQ1=HP9816 CONTROLLER
..SY=EQ2=HP3571 PRINTER
..SY=EQ3=HP3478 DVM
..SY=EQ4=M500AUSS HI-POT TESTER
..SY=EQ5=HP6034 POWER SUPPLY
..SY=EQ6=6" DIAL CALIPERS
..SY=EQ7=
..SY=EQ8=
..TR=/WI/HEADER.BP
..*
..** SET VARIABLES FOR PARTS MARKING
..*
..SY=ITEM1=A3011234-1
..SY=ITEM2=25.6 V  300 OHMS
..SY=ITEM3=10 AMPS AT 28 V dc
..SY=ITEM4=CIRCUIT DIAGRAM
..SY=ITEM5=TERMINAL IDENTIFICATION
..SY=ITEM6=SOURCE CODE   DATE CODE
..*
..TR=/WI/MARKING/PARTS.BP
..EX
```

Figure B

accomplished using EDIT.

INCOMING MATERIAL MANAGEMENT

Prior to installing an HP1000 computer in our Purchased Material Inspection department, incoming material management was a very labor intensive effort. A simple question from the Material Control Department : "Is this part in the inspection area?", required a search of the various shelves where the specific types of material would normally be stored awaiting inspection. Material deviating from the normal flow would complicate the task further. Additionally, the Inspection manager had no efficient or accurate method of measuring the current inspection work load. The quantity of incoming and outgoing boxes of material gave a vague measurement of throughput; however, the manager had no way of knowing the work load at each inspection station except to inventory the boxes at each station.

ATA was contracted to help solve these problems and streamline the flow of accepted and rejected material through the Inspection Department. The goal was to be able to easily use a computer terminal and access material locations and status of inspections.

ATA implemented several modular applications programs, intended to perform various tasks in a "friendly" environment, thus allowing non-computer oriented personnel to collect and report inspection information. The plan was to collect data at specific points in the inspection process;

1) - Receipt of material :
   - assignment of material to a station,
   - monitor station workloading,

2) - Completion of inspection :
   - collection of inspection results,
   - tracking accepted material to stock,
   - tracking rejected material to the "HOLD" area,

3) - Final disposition of rejected material :
   - Tracking material being returned to the vendor,
   - Tracking material being reworked,

Data is collected on data entry screens using HP FORMS/1000 and stored in a "Work In Progress" IMAGE data base. A second menu driven application was created to allow users to run various reports in batch at lunch or during breaks. Four reports were designed initially as follows;

- Part Status report - to report status of all lots of a given part number in the inspection area.

CURRENT PART NUMBER STATUS
MAY 28, 1985 8:58 AM

PMI INQUIRY FOR A3012701-1
TRANSISTOR, REV-B

LOT NUMBER	ASSIGNED LOCATION	A/R	OMR#	PMI COMPLETION REQUIRED	EXIT PMI
502076	A			85/05/18	/ /
502068	A			85/05/18	/ /
502024	A			85/05/17	/ /
502000	A			85/05/16	/ /
501783	A			85/05/03	/ /
501650	A			85/04/20	/ /
500707	HOLD	R	CR1232	85/02/18	85/03/09
402564	HOLD	R	CR1237	84/11/15	85/03/09
500773	HOLD	R	CR1233	85/02/25	85/03/09
500357	HOLD	R	CR1236	85/01/19	85/03/09
500437	HOLD	R	CR1241	85/01/25	85/03/11
500110	HOLD	R	CR0968	85/01/06	85/03/11

- Priority Report - to report all material waiting to be inspected sorted by length of time in the inspection department.

PMI LOT PRIORITY REPORT PAGE 3
MAY 28, 1985 8:08 AM

ENTER PMI DATE	LOCATION	PART NUMBER	LOT NUMBER	QTY RECEIVED (PMI)
85/05/02	CMM	A3013335-1	501836	36.000
	MECH	A3013335-1	501833	24.000
85/05/03	B	A3012699-1	501885	500.00
	D	A3012983-1	501892	700.00
	F	A3012999-1	501845	257.00
	HOLD	A3012711-1	501840	17.000
		A3012762-2	501854	0.0000
			501853	337.00
85/05/04	HOLD	A3013607-1	501887	184.00
85/05/06	HOLD	A3012711-1	501993	65.000

- Station Backlog Report - to report all material at each inspection station.

CURRENT PMI BACKLOG REPORT
MAY 28, 1985 7:55 AM

LOTS IN	PART NUMBER	LOT NUMBER	PART DESCRIPTION	QTY RECEIVED (PMI)	PMI COMP REQUIRED
CMM	A3013297-001	502101	HEATSINK, REV-H	40.000	85/05/23
	A3013306-1	502191	CHASSIS, REV-G	59.000	85/05/26
	A3013314-001	502144	HOUSING, REV-F	118.00	85/05/25
	A3013325-1	501836	CASE, REV-G	36.000	85/05/05
	A3013330-001	502212	CASE	18.000	85/05/31
	A3013330-001	502128	CASE, INCLUDING	24.000	85/05/24
	A3013333-001	502098	HOUSING, INCLUDI	56.000	85/05/23
	A3013335-001	502206	HEATSINK, REV-F	248.00	85/05/27
	A3013408-1	502127	FRAME, REV-C	23.000	85/05/24

TOTAL LOTS: 9

- Rejected Material Report - to report all rejected material and length of time waiting for disposition.

DMR ACTIVITY PAGE 1
JUNE 4, 1985 11:24 AM

ASSIGNED LOCATION	PART NUMBER	LOT#	DMR #	START DMR DATE	EXIT PMI DATE
COMP	A3012628-11	402834	CR0822	84/12/07	85/05/23
	A3012694-1	502015	CR1464	85/05/15	85/05/30
	A3012797-1	502259	CR1503	85/05/30	85/05/30
	A3012929-1	500270	CR0963	/ /	85/05/28
	A3012932-1	402984	CR0760	84/11/30	85/05/29
	A3012938-1	502273	CR1502	85/05/30	85/05/30
	A3013080-1	501748	CR1383	85/04/25	85/05/20

Users can also create custom reports using the IMAGE access language QUERY. Figure C shows the information available for reporting from the PMI data bases.

PMI DATA BASES

WORK IN-PROGRESS DATA BASE

INFORMATION AVAILABLE

- -Part Number
- -PMI Lot#
- -Location in PMI
- -PMI mfg code
- -Date Received in PMI
- -Qty Received in PMI
- -Sample Plan
- -Sample Size
- -Total # failed
- -Accepted/Rejected
- -DMR#
- -Date Exited PMI
- -Date on HOLD
- -Qty to Stock
- -Qty to RTV
- -Qty to Repair/Retest
- -Qty to Scrap
- -Qty to Other

NOTE: This information is currently available during the period from date received until 2 weeks after PMI exit.

QUALITY SYSTEM MANAGER DATA BASE

INFORMATION AVAILABLE

- -Part Number
- -PMI Lot#
- -PMI mfg code
- -Date Received in PMI
- -Qty Received in PMI
- -Sample Plan
- -Sample Size
- -Total # failed
- -Accepted/Rejected
- -DMR#
- -Date Exited PMI

NOTE: This information is currently available for the last 10 lots of a given part number from a given vendor.

Figure C

VENDOR QUALITY CONTROL

A vendor's "quality rating" is calculated based upon the ratio of unacceptable parts to acceptable parts in samples of past inspection lots. This quality rating then becomes an integral factor in the determination by Purchasing of

which vendor shall receive an order for the part in question. It should be noted that vendors may be evaluated on an individual part number basis, or on a cumulative part numbers submitted basis, as the database is capable of accommodating either inquiry. Figure D is an example of an inspection history for material from a specific vendor. This information is used when determining the sampling plan for material received, and is indicative of the information from which the quality rating is derived.

M.I.R.

PART NUMBER: A3012808-115 MANUFACTURER CODE: CE95^1

PART DESCRIPTION: CAPACITOR

PMI ENTRY DATE	LOT#	SAMPLE PLAN	PMI QTY RECEIVED	SAMPLE SIZE	NUMBER FAILED	ACCEPT REJECT	DMR#
85/05/15	502073	NORM	7952	200	0	A	
85/04/25	501743	NORM	6420	200	0	A	
85/04/18	501671	NORM	736	80	0	A	
85/03/06	500998	NORM	13166	315	0	A	
84/12/06	403092	NORM	17117	315	4	A	
84/11/28	402909	NORM	3085	125	0	A	CR0789
84/10/25	402454	NORM	10064	315	0	A	CR0530
84/09/27	401970	NORM	1011	80	0	A	
84/08/15	400525	NORM	9010	200	0	A	CR0111
84/08/07	400425	NORM	12530	315	2	A	CR0119

Figure D

The inspection environment is one which exhibits a high degree of both automation and parametric data collection. Data is accumulated at individual work stations using microcomputers and is transferred to the HP1000 via RS-232. ATA's applications software then reduces the raw data to summary statistics which are retained in the database, and histograms which are maintained in the vendor's file. These histograms may then be used to analyze a vendor's process capability, or to identify the drifting of a parameter toward a specification limit, as successive lots are compared to each other. A common example of this phenomenon is where a vendor's tooling wears, impacting a part's physical dimensions. The intent of this "trend" analysis is to identify potential problems in PMI before they become problems influencing the production of deliverable items. Figure E represents typical histograms available to PMI engineers for material analysis. Field engineers may access information which resides in the HP1000 by using portable terminals. These engineers request information regarding:

(1) Current revision of specification control documents,

(2) Status of vendor's lots currently in the inspection area,

(3) Failure modes of material submitted previously so that it is possible to discuss corrective actions on a timely basis.

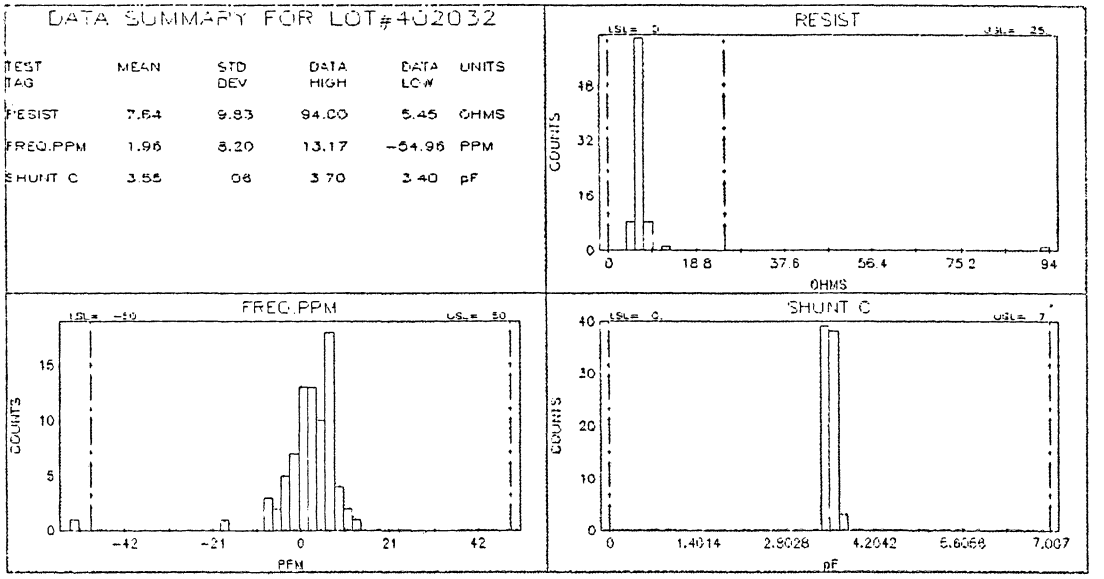


Figure E

CONCLUSION

All people at ITT's Aerospace/Optical Division, who have been involved in the justification, procurement, and applications design of PMI's HP1000, have experienced a great deal of satisfaction with respect to the extensive utilization of the system. The HP1000 has proven to be a versatile tool, providing written reports and on-line information which is used by several departments in the facility. The system has provided vendor quality ratings to the Purchasing department for use when procuring material. Material Control personnel depend upon the system to provide status of material which is required on the production floor. The Quality department, of course, has used the system to automate the generation and maintenance of inspection instructions, the tracking of material through the inspection process, the maintenance of inspection results, and the analysis of parametric test data.

1021. GETTING ACQUAINTED WITH THE 8-CHANNEL MUX

Carl A. Falstrom
ACCESS Corporation
4815 Para Drive
Cincinnati OH 45237

INTRODUCTION

If you use an HP 1000 computer and need to communicate with more than one or two serial devices, it is likely you will wish to use a product called the 8-Channel Multiplexer. (There was a 16-channel multiplexer which never quite achieved legitimate status within HP as a product, HP literature not withstanding. This paper will not deal with this earlier device.)

Brought out in two versions, one for M/E/F CPUs and one for A/L series CPUs, the two products are architecturally very similar, though electrically and mechanically distinct. This paper discusses the A/L series product, the HP 12040. However, from time to time, similarities and differences in the counterpart for the M/E/F family, the HP 12792 will be pointed out. The 8-Channel MUX is a powerful and flexible product, capable of supporting a wide variety of RS-232 peripherals, and to describe all of the features of the 8-Channel MUX is beyond the scope of this paper. Instead, an attempt will be made to convey sufficient information so as to enable the programmer to deal with typical problems he or she might encounter when interfacing a simple serial device such as a "Dumb CRT" to an A series CPU through a port on the 8-Channel MUX. A few pitfalls to be avoided will also be brought out.

While the bulk of this paper will deal with programming aspects of the MUX, a brief description of the hardware is necessary to understand some of the discussion to follow. The HP 12040 8-Channel Asynchronous Multiplexer provides a method for interfacing HP 1000 Computers using the A/L series backplane to any RS-232-C device. Device communication to and from the host system is provided through a microprocessor based interface. This card utilizes a Z-80A microprocessor in conjunction with EPROM, RAM, DMA and SIO support to manage the asynchronous serial protocol of connected devices. Each of the 8 channels is separately buffered (two 254 byte buffers for receive and two 254 byte buffers for transmit).

Each MUX contains two programmable baud rate generators (BRGs) which provide channel transmission speeds from 50 to 19.2K Baud. The MUX provides the ability to support five different baud rates to external devices simultaneously.

DEFINITION OF TERMS

Before attempting to come to grips with using the MUX, there are a few terms that are either unique to RTE-A or HP serial communication that should be defined first. Particularly important concepts to understand are device and interface drivers, the control word, ASCII and binary transfers, transparency mode, status (there are several kinds) and finally the distinction between end of data transfer and read termination.

In the A series architecture, there are two kinds of software drivers: interface

drivers and device drivers. Interface drivers are associated with interface cards (in this case, the interface driver IDM00 handles the 12040 MUX card). Device drivers are associated with peripheral devices attached to the cards. The 8-Channel MUX can use one of two device drivers for each channel, DD.00 or DD.20. DD.00 supports display terminals, such as the HP 262x series, or printers such as the HP 263x or 293x series. DD.20 is the cartridge tape unit (CTU) device driver to handle devices containing CTUs (HP 264x terminals). The same terms are used for the driver combinations used for the 8-Channel MUX on the M/E/F series of processors but the I/O architecture is quite different, and of course, the drivers have different names.

Device drivers DD.00 and DD.20 are designed to work with the interface driver for the HP 12005 ASIC card, ID.00, as well. While the characteristics of ID.00 and IDM00 are similar, enough differences exist for confusion to arise since the documentation generally describes use of DD.00, for example, with ID.00. Where important differences exist, I will try to cover them.

Interface drivers usually process I/O requests from device drivers. Device drivers process requests from application programs and pass them on to the interface driver. However, the device drivers for the 8-Channel MUX are written to support the specific characteristics of HP peripherals. If you must talk to a non-HP device, you may find it convenient or even necessary to bypass the device driver with your application program and make requests directly to the interface driver. This is fairly straightforward, and shortly we shall examine how to do this.

In the process of performing I/O, the interface driver and RTE share access to information in what is called an interface table (IFT). The interface table is constructed at system generation time, and there is one interface table for each MUX card installed in the system. The interface table for the MUX (as of A.85) is 9 words long, and in addition, there are an additional 20 words of IFT extension for storage of all temporary data associated with the MUX. For the curious, a description of the usage of the IFT and IFT extension words is given in Figure 1.

The generator also constructs a device table (DVT) for each device in the system. Each channel or port of the MUX must have its own device table, so there are 8 DVTs for each MUX. RTE uses the DVT for storage for its concerns, such as status indicators and list linkage words. The device driver uses the DVT for storage of device dependent data and as a communication area to the interface driver.

The DVT is followed immediately by the driver parameter area, or DVP. The contents of the DVP are defined at generation time, and this 12 word table supplies the device driver with information specific to the attached peripheral, such as whether it should be given the characteristics of a line printer, or a display. (In contrast, the device driver to be attached to a channel on the MUX on the M/E/F series may be changed with an EXEC control call to suit the peripheral.)

Immediately following the DVP area is the DVT extension area, DVTX. This 57 word table area is used by DD.00 and DD.20 for temporary storage and there is little in it of interest to the application programmer.

Areas of interest in the DVT/DVP/DVX area for DD.00 are illustrated in Figure 2. The driver parameter area is as it would appear if the device being interfaced to this MUX port were a 262x display terminal.

The control word (usually abbreviated CNTWD) supplies information to RTE identifying the device and supplies control information related to the I/O operation to be performed. The low order six bits, 0-5, identify the device LU. The meaning of the remaining bits, 6-15, depends on whether the device driver or interface driver is being called, and whether the I/O operation is a read, write, control, or status request. Most of the control bits affecting the MUX will be discussed later, but some of the I/O parameters controlled by these bits will be defined now.

Data transfer to and from most serial devices may be accomplished using DD.00 and IDM00, and the use of one or both of these drivers is assumed for the duration of this paper. The following definitions are specifically applicable for DD.00.

The Device Driver DD.00 supports writes and character reads in both normal ASCII and transparent ASCII modes. (Block mode keyboard reads are also supported, but this subject will not be discussed here). A normal ASCII write simply means that the device driver will add a CR/LF pair to the end of the output string, unless the last character in the string is the underscore. In this case, the underscore is not displayed and the CR/LF pair is not transmitted.

A transparent ASCII write does not supply the CR/LF automatically. It is the responsibility of the application programmer to supply these characters to the buffer where needed. The underscore character is also output to the display, even if it is the last character in the buffer.

During a normal character mode ASCII read special characters (e.g. DEL and BS) are processed and removed from the data stream. The data transfer stops with the occurrence of a record terminator. Record terminators recognized by the MUX are CR, control D (EOT), control R (DC2) or control ^ (RS). The record terminators, or delimiters, end the data transfer, but they are not passed through to the user buffer. If no record terminator is encountered, then data transfer is ended upon reaching the character count or word count specified in the EXEC call that originated the read request.

The transparent character mode ASCII read is very similar to normal ASCII reads. All the record terminators listed above except CR are not processed by DD.00, but instead are passed directly on to the user buffer. Data transfer ends when a CR is encountered, or on character count. In this mode, IDM00 behaves differently from ID.00. If the user buffer fills before the CR, the ASIC terminates the request immediately. The MUX, however, does not terminate the request unless it is specifically configured to terminate on character count.

For normal binary write requests the driver ignores processing of special characters. None are added or deleted from the user buffer. As far as IDM00 goes, there is no difference between a normal binary or a transparent binary write request. The device driver, DD.00 does not perform binary writes, but rather treats such a request as a transparent ASCII write.

During normal binary reads, special characters are not processed or removed.

The driver ignores CRLF processing and passes these characters to the user buffer as normal data. Data transfer ends when the character or word count is reached. IDM00 makes no distinction between transparent binary reads and normal binary reads.

The important fact to remember is that DD.00 is not capable of performing binary I/O. If you wish to do this, you must bypass the device driver and perform your I/O through IDM00. More will be said about this later.

There is a mysterious term mentioned in the 12792 MUX user reference manual called "honesty mode". This term is nowhere defined in the manual. Supposedly, bit 7 of the CNTWD establishes this, but for every form of I/O transfer listed that the MUX can perform, bit 7 is defined as a "don't care" bit, and the drivers ignore the state of this bit. My personal theory is that honesty mode is left over from the days when DVR00 was used with a paper tape reader. If bit 7, then called the V bit, was set, then DVR00 expected the first character read would indicate the remaining number of characters to be read less 1. Any other explanation for the use of this term would be welcome. In any event, the usage of bit 7 with the A/L series MUX will be dealt with shortly.

One other point of possible confusion to clear up before getting down to the business of using the MUX is the distinction between ending the data transfer and terminating the read request. The terms are used interchangably throughout the DRM, but they are not synonymous. Differentiating between the terms is only necessary for read requests. When a read request terminates, the calling program is rescheduled, and the data transferred, if any, is available in the user specified buffer, ready for processing. In contrast, ending a data transfer means only that the driver stops transferring incoming characters into the user buffer because the number of characters to be transferred as specified in the EXEC call has been reached. The read request is still active on the port, though, and the calling program remains suspended until the read request actually terminates, either because a record terminator character is recognized by the MUX or the 254 byte buffer on the MUX card becomes full.

The 8-Channel MUX is capable of supplying a multitude of information about its current status. The simplest information to obtain is the status of the previous EXEC call to the MUX port of interest. For lack of a better word, let's call this 'ordinary' status. To obtain ordinary status it is necessary either to examine the contents of the A and B registers or to issue an EXEC 13 call after an I/O transfer.

The information returned after this type of status call may not be sufficient. For an example, there is only one bit to indicate that an error occurred, but no information is given as to what kind of error it was. Now the second type of status, extended status, comes into play. After every I/O operation the driver updates the associated DVT with information about the I/O that just took place. The words that are usually of interest are DVT16-DVT18. How you may access that data and interpret it will be discussed in a later section.

The important thing to note about both types of status mentioned so far is that the information supplied is only valid after an I/O call has been made. This is because the data is being extracted from the DVT. The driver is not called.

There are situations where it is desirable to check the status of a particular MUX channel before performing any I/O, so that you can have reasonable assurance that the I/O transfer can successfully complete. The solution to this type of problem is called a dynamic status request. First, a control call with a function code of 6B must be made to the driver, which in turn interrogates the MUX card for status information. This "dynamic status" information is then stored in DVT16-DVT18 to be accessed by the programmer in a manner soon to be described.

GET READY, GET SET, ...

Arm yourself with a generator listing, the HP Driver Reference Manual (DRM), the 8-Channel Multiplexer Installation and Reference Manual, phone numbers for a friendly SE and CE and prepare to set up a MUX port for connection to a serial device.

The first thing that must be done before it is possible to issue a read or write request to a port on the MUX is to set up the Port ID. If you attempt to perform I/O to a port without setting up the Port ID first, your requests will be totally ignored by IDM00. You won't receive any error indication, but you won't accomplish anything either. The Port ID is set up by an EXEC control call with a function code of 30B. In Fortran, this looks as follows:

```
CALL EXEC(3,3000B+LU,PRAM1)
```

You can also issue the control call interactively from CI by entering

```
cn lu 30b pram1
```

This control call not only establishes a logical connection between the LU and the peripheral device connected to the port, but it also sets up eight parameters that are needed to define the basic communication characteristics of the port (e.g. baud rate, parity, handshaking, etc.) to match those of the peripheral. All 16 bits of the control call parameter PRAM1 must be set exactly right, or not only will you be unable to talk to the device on the port you are interested in, but you stand a very good chance of rendering most of the other ports on the MUX unusable as well. This can make you very unpopular with other users on the system.

Before proceeding, this seems like a good time at which to bring up a point about the software support for the 8-Channel MUX. At least one, and possibly as many as half a dozen control calls are necessary before you can perform a read or write to one port. Many of the control calls pack a lot of configuration information into the call parameter. Even a seasoned systems programmer may be seen balancing the Driver Reference Manual on one knee while he scribbles ones and zeroes in the margin and arranges them in groups of three to figure out the proper octal value for the control parameter. As has already been noted, consequences of an error in this process range from bizarre to disastrous. A "user friendly" MUX configuration utility is practically a necessity. I have submitted mine to the CSL and I imagine many other people have written software tools to remove the pain from configuring the MUX. Perhaps the best features of all of them could be combined into an HP supported program.

Back to setting up the Port ID. The first parameter to be defined is the number

of bits per character. This usually does not present much of a problem for the user since there are only four choices and it is only necessary to use the same number of bits that the peripheral expects. Of course, if you don't get it right, all you will see on a display or printout is unintelligible gibberish.

The next bit is used to indicate whether the port is to be used as a modem LU. It is at this point that the user may begin to have the first feelings of anxiety. The Driver Reference Manual contains a cautionary paragraph about use of this bit that is almost inscrutable. Even if a Systems Modem is eventually going to be employed, it is wise not to set this bit at first. Once a successful configuration has been achieved and tested with a hardwired connection to the peripheral one can then proceed to getting things working with the Systems Modem. This paper will not deal any further with the topic of the Systems Modem. Perhaps next year's conference.

The next parameter is the number of stop bits to be employed. I suppose there are some users who operate peripherals at 110 baud, and even fewer who need 134.5 baud (does anyone remember what a 2741 is?), and therefore require 1.5 or 2 stop bits, but 99.44% of the RS232 devices in use today work fine with 1 stop bit.

Next to be decided is which baud rate generator should be selected. Only one bit is involved, but this decision is by no means easily made. Earlier I stated that the MUX hardware has two baud rate generators. Depending on when you purchased your MUX, it is likely that either Port 0 is wired to baud rate generator 0 and Ports 1-7 are wired to BRG 1, or Port 7 is wired to BRG 0 and the rest of the ports are tied to BRG 1. Other wiring options are available from HP. (You can also rewire the connector to the MUX interface yourself and connect any port to either BRG, but if you are good enough to do this then you don't need to spend your time reading this paper.)

Assuming you know how your cable is wired, the next thing to be determined is the vintage of your MUX. If you have a 12040A, things are relatively simple. Two baud rate generators yield two baud rates. All ports wired to BRG 0 must operate at one baud rate, and all ports connected to BRG 1 must also operate at one baud rate, though this baud rate need not be the same as BRG 0.

If you have a 12040B or C, however, a clever twist has been added. Firmware changes allow programmatic control of a prescaler on the Serial I/O chips on the MUX card to provide three groups of baud rates for each BRG. As long as you make sure to select baud rates from only one group per BRG, you can program ports connected to the same BRG to operate at different baud rates (up to a maximum of three, depending on the group selected). Is all this perfectly clear? Details on baud rate grouping may be found in the 8-Channel MUX Installation and Reference Manual.

Ponder the setting of the BRG selection bit in the control call very carefully, because if you get it wrong and you issue the control call on a multi-user system, you might be unfortunate enough to change the baud rate that six other people are currently using to your desired baud rate. This sort of selfish behavior can damage relationships with your peers.

Now a selection of parity must be made. No problem: odd, even or none, right?

Well, almost right. There are two kinds of "none" listed in the Driver Reference Manual. Choose one and the parity bit will always be set to a 1, choose the other and the parity bit will always be set to a zero. Unfortunately, the proper choice can be crucial, and the DRM doesn't indicate which is which.

The gotcha here is that selecting no parity checking doesn't mean that the parity bit will always be ignored when the port is receiving data as you might expect. The handshaking and delimiter characters (more about these later) must have their eighth bit set exactly as defined by the Port ID configuration or the MUX will not recognize them. The symptoms of this kind of problem can be baffling.

During transmission, if handshaking is enabled, the MUX will send an ASCII ENQ character to the peripheral, expecting the peripheral to return an ASCII ACK character indicating that it is ready to receive data from the MUX. If the MUX expects the eighth bit to be a 1 and it is a 0, or vice versa, the MUX fails to recognize the ACK and the port 'hangs' indefinitely unless a time-out has been set. When receiving data, the message delimiter, the carriage return, for example, is not recognized as such. The MUX treats the delimiter as an ordinary data character, passes it through to the user buffer and fails to terminate the read request. No error is reported. So be careful when you specify "no parity" to choose the correct kind. The state of bit 9 in the control parameter is the state of the eighth data bit generated by the MUX during transmission and expected by the MUX during reception.

This brings up another point. If you do select parity generation and checking, and a parity error occurs during message reception, your read will be aborted with the dreaded "Transmission error occurred" message. Any data on the MUX card is flushed and the transmission log is set to zero. Isn't this treatment a little harsh for a simple parity error? It would be nice to allow the user the option of receiving a garbled message if he should so wish.

Well, we still have a way to go before we are finished setting up the Port ID. I have already mentioned ENQ/ACK handshaking. The state of the next bit in the control parameter turns this feature on or off. If the peripheral you wish to talk to is manufactured by HP, it will be compatible with this form of handshaking. Use it, and you can avoid data loss due to exceeding the ability of the peripheral to handle the data rate. But be sure that the peripheral is configured to perform ENQ/ACK handshaking or your port will hang when transmission is attempted. If the peripheral is not made by HP or designed to emulate an HP product, odds are that some other form of data traffic pacing is used, so do not enable ENQ/ACK handshaking.

The next four bits of the control parameter select the baud rate. Obviously you must match the baud rate of the peripheral, keeping in mind the previously mentioned idiosyncracies of the two baud rate generators on the MUX interface card.

Finally, we have arrived at the last three bits left in the control parameter. These bits define which of the 8 ports on the MUX is to be associated with the LU specified in the control call. This relationship has already been established at generation time, but now you are given the flexibility to change it. Some may regard this as a wonderful opportunity, but my experience has been that the necessity to select the port number in the control call just increases the risk

of earning the disfavor of your colleagues.

Once a successful 30B control call has been issued, you have met the minimal requirement to begin performing I/O to the peripheral. Many applications require additional control calls to set up other MUX functions, but discussion of these will be deferred until later in the paper. Let us now deal with transferring some data.

READIN', WRITIN' AND NO 'RITHMETIC

The Fortran calling sequence for a read request from a MUX port is:

```
CALL EXEC (1,CNTWD,BUFR,BUFLN,[PRAM3,[PRAM4]])
```

The parameter definitions for BUFR and BUFLN are the same as all HP read and write requests. BUFR defines the user buffer that is to receive the data, and BUFLN is an integer variable that defines the number of words to be received if it is positive, or characters to be received if it is negative. The meaning of PRAM3 and PRAM4 is dependent on whether the read request is being made through DD.00 or direct to IDM00. Shortly I shall deal with each case, but first let's examine the control word, CNTWD. The format of CNTWD is given below:

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 ------------------------------------------------
|BB|NB|UE|Z |WW|TR|XX|EC|YY|BI|    Device LU     |
 ------------------------------------------------
```

Bits 0-6 and 8-10 have basically the same meaning for the A/L series and the M/E/F series MUXes so I shall define these bits first. The main difference in the meaning of these bits on the two machines lies in the documentation. The mnemonics chosen for the 12792 product are inexplicably obscure, so I shall use the mnemonics from the 12040 manual.

The low order 6 bits (0-5) define the LU to be associated with the read request. (This limitation of 63 LUs plus LU 0 for the bit bucket has been addressed with the introduction of the XLUEX call, but I shall use the standard EXEC call throughout this paper.)

If bit 6 (BI) is set then a binary read is performed, and if bit 10 (TR) is set then transparency is in effect. This is only true for IDM00, however. It is important to note that DD.00 treats a request for binary I/O (i.e. Bit 6 set in the CNTWD) as a request for transparent ASCII I/O.

If a true binary read is desired, a way to accomplish this is for the application program to access IDM00 directly. To do this, it is only necessary to set the device driver bypass bit BB . Then to complete the process, you must set bits 14 and 15 in PRAM3 to 00. This will cause data transfer to end and the read to terminate when the user buffer is full.

If bit 8 (EC) is set, then echo is in effect. This simply means that the MUX firmware will immediately transmit back to the peripheral any characters it receives. If the peripheral is a terminal configured for full duplex operation, this is how the operator can see what is being typed. If echo is not desired, say for entry of a password, then the programmer may suppress it simply by

clearing bit 8 of the control word.

Bit 9 (XX) is interesting. This bit means different things to DD.00 and IDM00. Therefore, care must be taken when you use it. Bit 9 set to a 1 means that a program-enabled block read is enabled according to the DRM. However, this is only true if the terminal configuration, stored in DVP1, indicates page strapping is in effect (i.e. bit 3 is set to a 1). If bit 3 of DVP1 is a 0, then IDM00 interprets bit 9 as the "keep" bit. If KP is set, any data received by the MUX card that is beyond the end of the user buffer is saved and may be accessed by subsequent read requests. This allows you to "nibble away" at a message a piece at a time if you so desire. The usefulness of this feature is somewhat compromised by the inability of the MUX to pack multiple records into its 254 byte buffers. However, if you must manage large records and your buffer size is constrained, the "keep" bit may be an answer to your problem. In a little while I shall deal with how you may examine and modify the value in DVP1 to suit your purpose.

In M/E/F series systems, bit 7 is the mysterious "honesty" bit mentioned earlier. In A/L series systems, bit 7 (YY) is a multipurpose bit whose meaning must be defined in the context of whether or not DD.00 is being used, and the type of terminal to be employed with the LU specified at system generation time. IDM00 ignores the state of this bit entirely. On the other hand, DD.00 attaches special meaning to this bit only during write requests, so further discussion will be deferred to the next section.

The remaining bits in the control word have no counterpart in the M/E/F series MUX.

Bit 12 (Z) is useful if you have a need to perform two I/Os to a peripheral with a single EXEC call. An example of this might be to write a prompt message on a display and then read the user entry. If the Z bit is set, then PRAM3 must define the buffer containing the message to be sent to the peripheral and PRAM4 must define the length of the message. (The conventions employed by the BUFR and BUFLN parameters apply here as well.)

Bit 13 (UE) should be set if you don't want RTE to write nasty messages on user's terminals or the console when an I/O error of some sort occurs. By setting this bit, you inform the driver not to suspend the calling program, down the device or take any sort of gratuitous actions whatsoever. It now is incumbent on you, the programmer, to interrogate the status after each I/O request, and take what measures you deem appropriate. I shall discuss how to examine status shortly.

Setting bit 14 (NB) normally means that the I/O transfer is not to be buffered by RTE. However, EXEC ignores the state of this bit during read requests and always performs a non-buffered read. This is generally desirable, because meaningful status returns after the read request completes are available. However, it does mean that the program is not swappable for the duration of the read request. If this is a problem, the way around it is Class I/O. More about this bit later when I discuss writing information to a peripheral.

I have alluded previously to the fact that it is possible to perform I/O directly through the interface driver, IDM00, without having DD.00 perform any processing on your request. All that it's necessary to do is to set bit 15 (BB) and this is just what happens. Easy, huh?

I have already discussed usage of PRAM3 and PRAM4 in the context of DD.00, and these parameters have no counterpart in the M/E/F MUX.

When calling IDM00 directly, PRAM3 takes on an entirely different meaning and must be supplied. PRAM4 is also redefined, but its use is optional.

Page 3-37 of the DRM defines the usage of these parameters with IDM00. Basically, bits 15-14 of PRAM3 define how the read request is terminated (the DRM merely states that the data transfer ends, but the read request terminates too) and bit 11 turns echo on or off.

The left hand byte (bits 15-8) of optional PRAM4 allows the programmer to specify a prompt character to be output before the read is started. If the value of this byte is zero, then no prompt character is output.

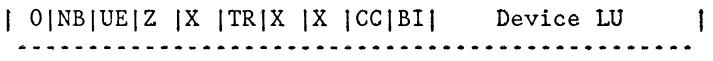
The utility of the low order byte of PRAM4 appears to be limited to rather special circumstances, but perhaps it may be just what someone reading this paper is looking for. If this byte is non-zero, then it defines the number of characters beyond the end of the user buffer that are to be discarded by the driver during a binary read.

By now you should be sufficiently well informed to handle all but the really nasty types of reads, such as variable length data that is not terminated by anything the MUX recognizes and the message length cannot be determined in advance. If you want to do this, be patient, and after a few more MUX features have been introduced, we will discuss this kind of problem. Meanwhile, let's move on to write requests.

In Fortran, you write to a MUX port like this:

```
CALL EXEC (2,CNTWD,BUFR,BUFLN[,PRAM3,PRAM4])
```

The definitions for BUFR and BUFLN are as usual. Defining the format of the CNTWD is slightly more complicated. The M/E/F series MUX uses a very different definition from the A/L series. I shall only discuss the latter and refer the reader to page 2-6 of the 12792B MUX User's Manual for details on the former. First let's look at the CNTWD when DD.00 is used:



Most of these bits should be familiar by now as they have the same meaning as in a read request.

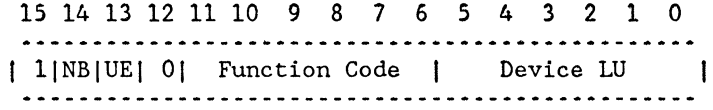
If the Z bit is set, PRAM3 and PRAM4 must be supplied. The definition of these parameters is the same as for a receive request.

Bits labelled X are "don't care" bits, although the DRM recommends that they be set to zero.

Bit 7, here labelled CC, only has meaning if the LU to be written to has been

defined at system generation time to be a hard copy terminal, (i.e. device type 12 in DVT6). In this case, if CC is set to 1 the driver uses single spacing and outputs the entire user buffer to the printer. If CC is 0, then the first character in the user buffer is not printed, but rather is interpreted as a carriage control character. The conventions for this character are the same as employed by most HP line printers. An ASCII '0' means double space, a '1' means eject the page, an '*' suppresses the line feed, and any other character is ignored (and not printed). If the LU has been defined as a display terminal, bit 7 is a "don't care" bit. Also, DD.00 will not interpret the CC bit in transparent mode.

Now let's examine the format for the CNTWD when DD.00 is bypassed and you wish to write directly with IDM00:



Double buffered I/O is performed by DD.00, so note that the Z bit is zero, and PRAM3 and PRAM4 should not be supplied. Bits 0-5 and 13-15 have the standard meaning, but something new has been introduced in bits 6-11. The meaning of the bits defined as of this writing is as follows:

Bits 6 and 7 are related. Basically, if either bit is set to a 1, IDM00 does not append a CRLF to the data in the user buffer. Only by setting both bits to 0 will you get the CRLF. No explanation is offered for why two bits are required to perform this simple function.

Bit 8 controls ENQ/ACK handshaking. If handshaking has been previously enabled with the control 30B call, it will be disabled for THIS WRITE REQUEST ONLY if bit 8 is set to 1. If bit 8 is 0, the handshaking takes place as defined by the 30B control call.

Bit 11 controls a mysterious function called the Non-Display Terminator (NDT) which is the ASCII string "ESC_". This is interpreted by 264x series terminals as an end-of-line termination. If bit 11 is set to 1, IDM00 outputs an NDT after printing the user buffer. Otherwise it doesn't.

Probably the most useful values for the function code are 0 for simple transmission of ASCII data, and 35B for binary transmissions. When a function code of 35B is specified, the user buffer is transmitted exactly as supplied - nothing added, nothing deleted. What happens when you try other combinations of bits 6-11 will be left as an exercise for you, the reader, to find out.

Now that we have covered reading and writing, let's see what you have to do if you wish to find out whether your program may safely attempt the I/O transfer, or if the transfer has already completed, whether it took place without error. This type of information is provided by the MUX driver when a status request call is made.

The simplest way to obtain status information after a read or unbuffered write request to a MUX port is to examine the contents of the A and B registers after

the EXEC call completes. If you are not programming in assembly language, the utility subroutine ABREG may be called to obtain the data. The A-register contains word 6 of the DVT, which is the device status passed by the driver. Refer to Figure 2 to see how to interpret the information in this word. The B-register contains the transmission log, which is the positive number of words or characters (depending on how BUFLN was specified in the original EXEC call) that were transferred.

Another way to obtain status is with an EXEC 13 call. The format of this call is given below:

```
CALL EXEC (13,CNTWD,STAT1,STAT2,STAT3,STAT4]]])
```

The format of the CNTWD is simple. As usual, bits 0-5 indicate the LU of interest. Bit 12 is the Z bit, which I shall define in a minute, and all other bits must be zero. STAT1 is an image of DVT6, defined in Figure 2. If supplied, STAT2 will contain an image of IFT6 which is defined in Figure 1. The meaning of STAT3 and STAT4 is dependent on the state of the Z bit. If Z is 0, STAT3 contains DVP1 and STAT4 contains DVP2. If Z is set to a 1, STAT3 defines a user buffer area that is to be passed an image of the DVP area, and STAT4 defines the length of this buffer.

Perhaps bit 0 of STAT1 is set, indicating some form of error occurred during the transfer. If you would like additional information about the specific problem, then you must examine the extended status information placed by the drivers in DVT16-18. This may be done by making a call to the utility subroutine RMPAR. The interpretation of these three words may be found in Figure 2.

The information we have covered so far should be sufficient to handle basic I/O transfers to and from a serial device attached to a MUX port. We shall now turn to the use of some of the more exotic features of the 8-Channel MUX. To use these, some more control calls must be introduced.

WHO'S IN CONTROL HERE?

The next few control codes to be discussed provide an additional degree of control over the MUX for applications that require it. Normally, the next call to be issued after the 30B call is the 33B call. Remember, the format for a control call in Fortran looks like this:

```
CALL EXEC (3,NN00B+LU,PRAM1)
```

where NN is the control function, in this case, 33B. For this and the control functions to be discussed soon, I shall not go into as much detail as I have with the 30B call. The DRM provides copious information for each control call. Instead, I shall limit myself to clarification of some of the information given in the manual and pointing out a few potential traps to avoid.

The important functions defined by the 33B control call are device error handling, type-ahead control, program scheduling, break key processing, and configuration of read requests.

Type-ahead, for those who are new to the 8-Channel MUX, is the ability of the

MUX to buffer received data if no read is pending and pass the data to the interface driver when a read request is made. It is generally a nice feature to use since it increases the apparent response time of the system but it can create some problems too. For a good discussion of the pros and cons of type-ahead, refer to pages 3-56 thru 3-58 of the DRM. Depending on how you configure PRAM1, you can disable type-ahead or enable it. You can also attempt program scheduling when type-ahead data is available on the card, or you can merely set a bit in the DVT to notify the user that type-ahead data is available.

Action to be taken when the user hits the BREAK key can also be defined. Normally, when type-ahead is used, the BREAK key is the only way to get RTE's attention.

A very important function performed by the 33B control call is to instruct IDM00 how to control sending of read configuration to the MUX interface card. Typically, read configuration information is passed to the MUX in bits 6-15 of the CNTWD of the read EXEC call as discussed earlier. However, there are times when it is desirable not to allow the read request to change the configuration. By setting bit 7 of the CNTWD to a 1, you can tell the interface driver to ignore the configuration bits in the read EXEC call. An example of this will be give shortly.

A word of caution concerning the 33B control call. If you have specified a printer, such as the 2631B, to be associated with a given MUX port at generation time, DO NOT issue a control call with a function code of 33B to this port. This can crash the system and necessitate a reboot.

If you decide to prohibit read configuration information in the read request from affecting the MUX, then there needs to be a way to tell the MUX what configuration to use. This is where the 37B control call comes in. PRAM1 in this call configures the MUX card for read requests. Remember, for this configuration to remain in effect, bit 7 in a previously issued control 33B must have been set. Otherwise, any read request will override these parameters.

The format for PRAM1 follows. When the specified bit is set to a 1 it enables the function:

```
15 - End transfer on carriage return (CR)
14 - End transfer on control ^ (RS)
13 - End transfer on control D (EOT)
12 - End transfer on control R (DC2)
11 - End transfer on character count (Set by a control 36B call)
10 - End transfer according to bits 15-12
 9 - Enable input data editing
 8 - Enable input data echo
7-0 - Reserved
```

Most of these functions are self explanatory. So far, I have not mentioned data editing. This simply means that when editing is enabled, the backspace and delete characters are not treated by the MUX firmware as data characters and entered into the user buffer. Instead, the backspace erases the last character received in the MUX receive buffer, and the delete erases the content of the entire buffer and transmits a backslash to the peripheral . Obviously, bits 10

and 11 are mutually exclusive, so don't set both of them in the same control call.

If you have set bit 11 in the control 37B call, then there must be some way to tell the card when enough characters have been received to end the transfer. The control 36B call accomplishes this. PRAM1 in control call 36B should be a positive integer in the range of 1-254 defining the number of bytes to be transferred by the read request.

So far we have dealt with control calls handled by IDM00. There are several more that we shall return to, but now we shall consider some frequently used control calls that DD.00 manages.

First, there is the control 20B/40B call for enabling the scheduling of a primary and secondary program. Primary and secondary programs to be scheduled for each MUX LU are usually specified at system generation time, and stored in DVP5-12. The names of these programs may be changed with these control calls. Similarly, a complementary pair of control calls, 21B/41B may be used to selectively disable scheduling of either the primary or secondary programs. It is also possible to disable scheduling of either program with a control call of 23B.

Frequently, it is desirable to set a time-out on a port so that a read request won't tie up the port indefinitely. Here is where a major departure occurs on the A/L series from the technique employed on the M/E/F series MUXes for detecting time-outs. On M/E/F systems, control call 22B sets a time-out on the port. Each port uses two words in the EQT for timing user requests. One word, if non-zero, contains the time-out value in tens of milliseconds, and the other word is used as a clock. Each time a read, say, is posted, the driver copies the time-out value into the clock word. Then, each time the TBG interrupts, any non-zero clock word has its content decremented by one. Should a clock word decrement all the way to zero, then the port has timed out and appropriate processing takes place.

This technique is used on the A/L series, but only for system calls to the driver. A control call of 22B establishes the system time-out for a given port and the technique used to implement time-out processing is essentially that of the M/E/F series.

A completely different method is used to handle user request time-outs. A control call of 27B must be issued to set a time-out for user requests. This causes the system to issue an ENQ to which the device must ACK within the specified time-out period. This is fine if the peripheral can deal with the ENQ/ACK protocol. Unfortunately, some devices react in unpredictable ways when they receive unexpected characters. There is no nice way I know of to set a time-out on a port that avoids this problem. Incidentally, the DRM says ENQ/ACK must be enabled in order for time-outs to work. I have found that if a port has a time-out set, the system generates ENQs no matter how control 30B has configured the port. I assume that the caution refers to the fact that the peripheral device must have ENQ/ACK enabled for time-outs to work properly.

As has been mentioned, DVP1 is set up at system generation time to specify the terminal configuration. Should you wish to change this information, DD.00 provides a control call (Function code 44B) with which to do it. The value of

PRAM1 is stored in DVP1. Handy.

There is another control call that DD.00 provides that is very useful for talking to non-HP devices. If you use an HP terminal connected to an A series CPU for program development and you turn display functions on, you will notice after the CI> prompt, a DC1 appears. Normally, this character is transmitted by the MUX prior to posting a read request on the port. The DC1 is used by HP terminals to trigger a read transfer. As in the case of the ENQ character mentioned above, many non-HP devices don't take kindly to unexpected characters being sent to them. The control 45B call can come to your rescue. With this call, you can modify the trigger character (which is defined at system generation time and stored in DVP3) to any other character by setting the desired trigger character in the upper byte of PRAM1, or no trigger character at all by setting this byte to zero.

Let's now return to some control codes handled by IDM00 that come in handy. A typical problem encountered when dealing with some serial terminals is that depression of the carriage return key after entering a line of text not only generates a CR but also a LF. This so-called auto-line feed may not be easy to disable. If type-ahead is enabled, here's what happens: The MUX recognizes the CR as the read terminator and does just what you would expect - it terminates the read. Then the LF comes down the line. A read is not now pending, but since type-ahead is in effect, the LF character goes into the active receive buffer on the MUX card and gets tacked onto the front of the next message. A way to avoid this annoyance is to perform a control 26B buffer flush call prior to issuing the read request to the port. This will clear any data that may have accumulated in the port's active input buffer while in type-ahead mode.

Function code 34B is useful for transmitting to peripherals that use the popular (outside of HP) Xon-Xoff traffic pacing protocol. If you have a B version MUX or later, you can enable or disable Xon-Xoff traffic pacing. Be careful if you enable this type of handshaking to avoid several traps: HP's EDIT/1000 uses the Xon (control Q) and Xoff (control S) for edit functions, and so does BASIC. Since the MUX will treat these characters as handshaking characters with Xon-Xoff enabled, you can have problems. It is also not advisable to attempt to read binary data with this form of traffic pacing enabled.

A problem can occur with this protocol should a peripheral send an Xoff to the MUX and then go down. After the peripheral is brought up, how is it possible to force it to send an Xon to allow transmission to resume? This is one of several problems addressed with the C release of the MUX. A new function has been added to the 34B control call which allows you to go to a working port and issue a command that fools the MUX into thinking it has received an Xon from the previously downed peripheral.

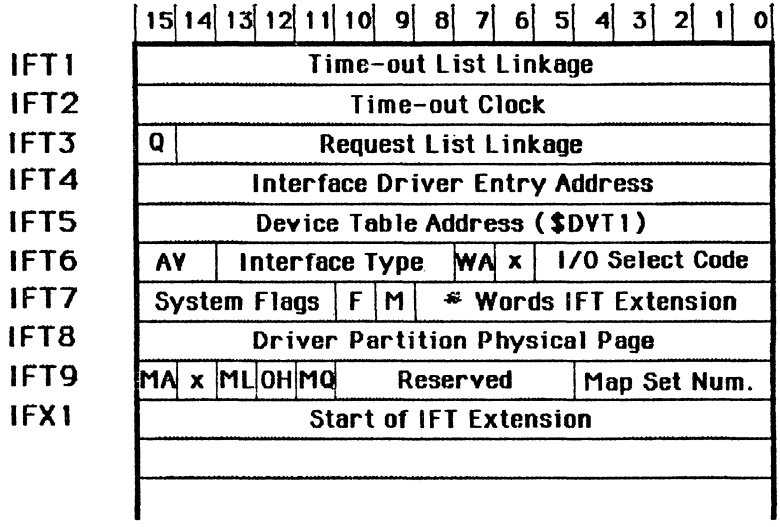
One last control call, and then we shall look at a nasty interfacing problem and how it can be solved. This control call uses function code 52B and its purpose is to immediately terminate the active receive buffer on the MUX card. A subsequent read request will read however many characters are on the card (you can find out what the transmission log is from the B register or by making a RMPAR call to retrieve the contents of DVT17). This call is useful for solving a problem I alluded to earlier, that of having to read variable length messages that are not terminated by anything recognized by the MUX. A program fragment

illustrating the use of this call is given in the DRM and I have reproduced part of it here:

```
C
C     SET THE PORT UP FOR TYPE-AHEAD AND DON'T RECONFIGURE
C     THE READ OPERATION ON A READ REQUEST.
C
           CALL EXEC (3,3300B+LU,22200B)
C
C     SET THE READ CONFIGURATION TO END ON A COUNT OF 254, ECHO
C     AND EDIT OFF
C
           CALL EXEC (3,3600B+LU,254)
           CALL EXEC (3,3700B+LU,4000B)
C
C     THE FOLLOWING LOOP WILL TERMINATE AND READ THE INCOMING
C     DATA BUFFER
C
10         CALL EXEC (3,5200B+LU,0)
           CALL EXEC (1,LU,BUFR,-254)
           CALL ABREG (ISTAT,LEN)
C
C     LEN CONTAINS THE TRANSMISSION LOG
C
                    :
                    :
C
C     PROCESS THE DATA
C
                    :
                    :
C
C     READ SOME MORE DATA
C
           GO TO 10
```

The program does work as advertised, but it has an unfortunate weakness that I have found no way to circumvent. For the program to work, it is necessary to set bit 7 in the 33B control call. Hence, IDM00 dutifully does not update the MUX card with information contained in bits 6-15 of the control word. This in itself is not a problem, because most of the information conveyed in those bits can be supplied instead from PRAM1 of the 37B call. Most of the information, that is, but not all. In particular, I did not want the driver to perform error processing, but there appears to be no way to inform the driver of this fact because the UE bit in the CNTWD is ignored. Perhaps some of the available bits in PRAM1 of the 37B call could be used to convey this sort of information to the driver when it cannot be supplied by the EXEC read. With the A.85 release, it is possible to prevent MUX buffer overflow errors from being reported by RTE by setting bit 4 of DVP1, but parity errors are still reported.

Hopefully, the information I have presented in this paper will be of use to some of you who have taken the trouble to stay with me to the end. If I can save one person from the hours of frustration I have endured learning the hard way, then I shall consider the effort well worth while.



Q - Queuing option
0 Requests queued by priority
1 Requests queued FIFO

AV - Availability Field
0 Interface available
1 Interface locked to DVT
2 Interface is busy
3 Interface is locked and busy

WA - Waiting to abort bit

F - First entry bit

M - List dequeuing control

MA - Map allocated

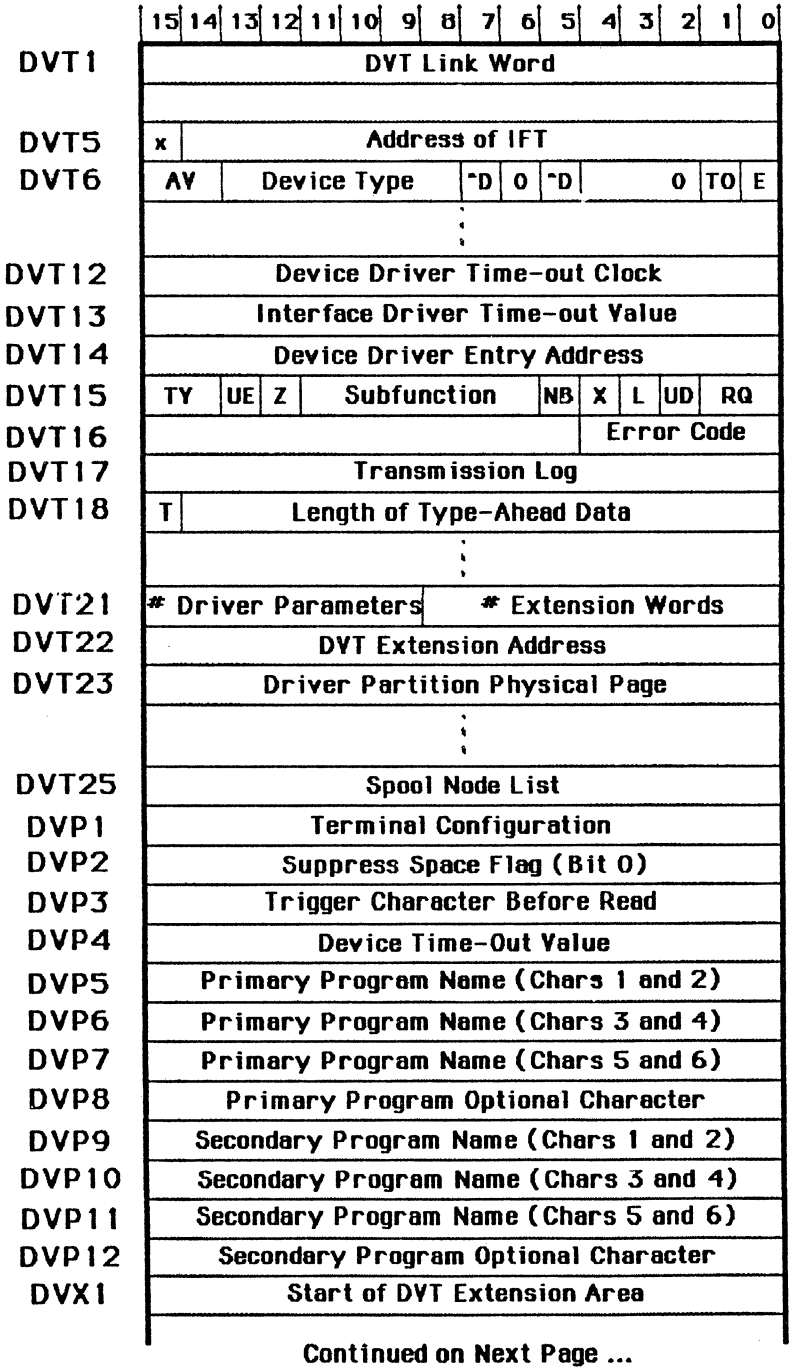
ML - Map Locked

OH - Waiting for map set

MQ - Map set queued

Map set number is the number of the map set allocated for this I/O channel. Only valid if MA is set to 1.

Figure 1. Interface Table Format



Continued on Next Page ...

AV - Availability
00 Device Available
01 Device Down
10 Device busy
11 Device down and busy with request

Device Type: 00,05,12

^D indicates that a control D was entered from the keyboard. Bit 5 indicates EOF and Bit 7 indicates EOM

TO indicates a time-out

E indicates an error. See DVT16 for explanation.

TY - Request Type
0 User Program Request
1 Buffered User Request
2 System I/O Request
3 Class I/O Request

UE - User Error Bit (from EXEC request).

Z - Double Buffer Bit

NB - Nonbuffered bit

L - Data Location bit

UD - Device Driver Bypass bit

RQ - Request Type
0 Multibuffered request
1 Read, Write/Read request
2 Write request
3 Control request

Subfunction is bits 6-11 of the EXEC CNTWD.

Error Code -
3 Timeout
5 Transmission error
77B Request aborted by RTE

T - Type-ahead data available

Figure 2. Device Table Format

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VX1	Address of current DVT															
VX2	DD.00 internal use															
VX3	DD.00 internal use															
VX4	DD.00 internal use															
VX5	RW	WB	BU	RP	x	PO	x	x	Port Key							
VX6	ND	RT		TA	PE	MD	SE	PK	Last read configuration							
VX7	Length of type-ahead data															
VX8	Pram 1 of control 33B call															
VX9	Pram 1 of control 30B call															
VX10	Pram 1 of control 34B call															
VX12	DVT address for program scheduling															
VX13	Temporary Storage															
VX14	Temporary Storage															
VX15	Temporary Storage															
VX16	Temporary Storage															
VX17	Pram 1 of control 31B call															
VX18	Pram 1 of control 32B call															
VX19	Modem alarm program (Chars 1 and 2)															
VX20	Modem alarm program (Chars 3 and 4)															
VX21	Modem alarm program (Chars 5 and 6)															
VX22	DD.00 internal use															
VX23	DD.00 internal use															
VX24	DD.00 internal use															
VX25	DD.00 internal use															
VX26	DD.00 internal use															

DVT Extension Area

RW - Read/Write to card
WB - Waiting for xmit buffer
BU - Internal buffer in use
RP - Request pending
PO - Parity/overflow int. stat.
ND - Not end of msg yet
RT - Last terminator recvd.
TA - Type-ahead data available
PE - Parity error/overflow
MD - Modem line down
SE - Scheduling enabled
PK - Port has key

The last read configuration is as set by a Control 37B call or the last EXEC read, depending on the state of bit 7 of Control call 33B.

Control calls 31B and 32B are for use with the Systems Modem Panel or the Modem Card.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DVP1	PM	AS	T	0									FB	SE	PS	LF	FF	EQ

Terminal Configuration Word

Terminal Configuration Word Format:

PM - Page mode bit
AS - ASCII bit (0=7 bits, 1=8 bits)
T - Termination bit
1 Terminate on count
0 Normal termination
FB - Flush input buffers on terminal status request
SE - Suppress transmission errors caused by buffer overflow
PS - Page strapping bit
LF - Line Feed bit. Must be set for IDM00.
FF - Form feed enable bit
EQ - Enq/Ack enable bit. Don't set when using MUX. Use CN 30B to control handshaking.

Figure 2. Contd. Device Table Format

1022. REAL TIME MANAGEMENT CONTROL OF FACTORY OPERATIONS

K. Benson
PCI Ltd.
Snaithing Grange
Snaithing Lane
Sheffield S10 3LF
South Yorkshire
England

INTRODUCTION

In the current economic climate, it is essential that all manufacturers maintain tight controls over both factory costs and their investment in stock and work-in-progress, while attempting to meet customer delivery commitments.

To enable the manufacturer to meet these objectives, he must have efficient pre-production planning routines, and when work is actually loaded into the factory, each production order should be controlled against a pre-planned timescale to ensure an acceptable level of delivery performance.

While many manufacturing processes have generally become much more complex to organise and operate in recent years, the tools provided to assist management and supervision in the control of factory operation have not changed dramatically. Little wonder, therefore, that the type of problems mentioned above are all too prevalent.

Although computer-based production planning and control systems have been available to industry for some years, they have typically reported historically on what happened in the factory the previous day, or the previous week - too late to take effective corrective action. If we are to control factory operations efficiently, front-line management and supervision must have access to key information as it happens, so that they can respond quickly to correct an imbalanced situation. This information can only be made available via a Real-Time system, such as the PCI Production Control System.

DEFINING THE SYSTEM

The PCI SYSTEM controls the following functional areas in a manufacturing company:

* production planning
* production scheduling
* factory loading
* factory progress control
* section balancing
* work-in-progress level control
* gross payroll and labour cost control

At the beginning of each manufacturing period, certain basic information is fed into the system such as:

PRODUCTION SECTION DATA: information on operators assigned to each section,

hours worked, operation(s) performed, supervisor responsible, etc.

OPERATOR PERFORMANCE HISTORY: details of regular operation(s) assigned, performance levels achieved, other operations worked on, etc.

PRODUCT SPECIFICATION DATA: for each product to be manufactured-operation numbers, descriptions, standard piece rate values, operation sequence.

When a production order is raised, information such as order number and quantity, planned issue date to manufacturing and planned complete date, is input to the system. Batch data is added - batch number and quantity - and the actual issue date to the factory is input, when the order is loaded.

FACTORY OPERATION

A small terminal, known as the Operator Input Device (OID), is located at each operator's workplace. Each operator has a clock card which contains bar-coded information. After arriving at his workplace at the beginning of a shift, he enters his clock card via the terminal, and in this way, the system knows that the operator has arrived for work and the time clocked is automatically read by the system.

Details of the operation to be performed by the operator at that workplace are input via an Operation Card, which contains details of the operation number (in bar-coded form) and operation description. This Operation Card is similarly read by the terminal. Now the system knows which operation the operator will be working on, at least for the first batch of the day.

Each batch of work has a Batch Card travelling with it, and the batch details - production order number, batch number, quantity and product number - are held on the Batch Card, again in bar-coded form. As the operator completes work on the batch, he detaches the batch card from the work and reads it through the terminal, into the system. The Batch Card is then returned to the batch of work, to be moved on to the next operation in the manufacturing sequence.

For each of these transactions, a signal is given to the operator to indicate whether the data has been correctly received by the system. If the data has been accepted, a single flash of the LED display and a single bleep from the audio device in the OID is provided; however, if the data is not accepted by the system, several flashes and bleeps are produced simultaneously.

When the operator has to go onto an off-standard category of working, such as machine breakdown, or waiting time, he logs off the system by reading his clock card through the OID. This action produces a warning message on the Hewlett-Packard Visual Display Unit (V.D.U), which is located at his Supervisor's workplace in the factory. This VDU is used by the supervisor to authorise all periods of operator off-standard time.

Again, each terminal entry is time stamped, so that the computer can calculate the actual time spent at off-standard for each category, for each direct operator. This information, together with the batch data, which is captured for all on-standard working, enables the gross pay to be calculated daily and weekly

for each direct operator.

All data collected from the factory floor is verified and edited by the PCI Data Concentrator, which is capable of receiving data from and transmitting data to a maximum number of 128 Operator Input Devices (O.I.D.s).

The Data Concentrator receives and checks data from the O.I.D.s and sends an acknowledgement back to the O.I.D.s concerned. Through the use of the Data Concentrator's own on-board microprocessors, upon receipt of a successful read, the data is transferred from one microprocessor (Z80 based), where the data is Date and Time stamped, is stored in memory and transmitted to the Hewlett-Packard computer for processing.

The Power Fail Recovery System provides the Concentrator with Battery Back Up to protect against failure of mains supply.

A Hewlett-Packard A600 Series machine is utilised as the system processor, and a 65 megabyte disk is employed as standard. The bar-coded cards are produced on a Hewlett-Packard 2934A Printer, and cut to size using a Guillotine, which is supplied as part of the system hardware.

SYSTEM OUTPUT

The output information generated by the system consists mainly of visually displayed data provided for the section supervision and line management. Some hard copy reports (largely for payroll/labour cost control purposes, and for production planning and scheduling routines), are produced on the printer. Reports are generated either automatically at pre-determined intervals during the day, or on request when immediate corrective action is required by the supervisor or line manager.

The applications software for a typical real-time system covers five major areas:

* PLANNING AND SCHEDULING REPORTS
  * Production Scheduling
  * Factory Loading

* PROGRESS CONTROL REPORTS
  * Works Order Status
  * Batch Status

* PRODUCTIVITY REPORTS
  * Dynamic Balancing
  * Production and Work-in-Progress

* PAYROLL/LABOUR COST CONTROL REPORTS
  * Pay/Performance Status
  * Payslip Print
  * Section Reports
  * Labour Cost Control

* INVENTORY CONTROL REPORTS

PLANNING AND SCHEDULING REPORTS

* PRODUCTION SCHEDULING

The System produces a Report which details, by Department, the works orders scheduled to be loaded for the next 'x' weeks, and compares this planned load, by week, against available weekly capacity, in units or standard minutes, to determine potential weekly over/under load situations, in advance of order issue.

* FACTORY LOADING

This report details, for a specific week ahead, by Department, the work content of the works orders scheduled to be issued to the factory that week, in key sections, machines or operations, and compares this planned load against available weekly capacity of those sections, machines or operations, in units or standard minutes, to determine potential local bottleneck situations, prior to releasing the order for manufacture.

PROGRESS CONTROL REPORTS

* WORKS ORDER STATUS

The system produces a report which shows the current status of any works order and its relative position against target completion dates, at key progress points down the manufacturing cycle.

* BATCH STATUS

A screen report is produced for the Supervisor which identifies overdue batches within a works order which have failed to reach a key progress point by the scheduled completion time necessary if the works order is to be completed on time.

PRODUCTIVITY REPORTS

* DYNAMIC BALANCING

The system is constantly checking actual production achieved per operation against daily targets set by management. When an imbalance situation is highlighted, a Dynamic Balance Report is produced on the Supervisor's screen. Production achieved day-to-date figures are detailed for the operation and also for the operators/machines currently assigned to that operation. In the case where actual production achieved is significantly below target, details of operators and/or machines to assign are shown, to assist the Supervisor to efficiently balance his section.

* PRODUCTION AND WORK-IN-PROGRESS

Details of production, in units and standard minutes, are produced for the day and for the week-to-date. These actual production figures are compared against targets set by management, and the variances expressed as a percentage. Information relating to the actual level of work-in-progress by Department is produced, which is compared with the target level and the variance figure is shown. The level of

work-in-progress is expressed in equivalent days production.

PAYROLL/LABOUR COST CONTROL REPORTS

* PAYROLL/PERFORMANCE STATUS
A dynamic report is provided for the Supervisor on demand, which details up-to-the-minute pay and performance figures for each operator in the Department.

* PAYSLIP PRINT
At the end of each working day, a payslip is produced for each operator, which details gross pay earned that day, broken down into on-standard and off-standard pay elements.

NB: These payroll-related Reports are only meaningful if the factory is operating an individual piecework payment system for the direct operators.

* SECTION REPORTS
Daily/Weekly Reports are produced, which detail at operator level, by Department:

* clock minutes worked - on standard/off standard
* SMs earned - on standard/off standard
* operator on standard performance
* operator pay - on standard/off standard
* department efficiency
* cost per standard hour
* average earnings per hour

This performance-related data is used for Supervisor follow-up and control.

* LABOUR COST CONTROL
This report, which is produced daily, details achieved figures, by Department and for the factory in total, for key control parameters, such as:

* on standard performance
* utilisation
* efficiency
* analysis of excess costs - by category
* production achieved (units/SMs)
* direct labour costs
* cost per unit produced
* cost per standard hour
* minutes per unit produced
* analysis of operator performance, etc

These figures are compared with budget, and the variance is shown. Daily figures are summarised into a weekly report.

* INVENTORY CONTROL

A wide range of Reports, both in the visual and printed format, are produced to handle all functions associated with raw material/bought-out component stock control, including:

* Order Analysis - provision of bill of material requirements from each customer order or contract;

* Full stock control facilities - processing of customer orders, purchase orders, material/component receipts, issues, returns, amendments and cancellations;

* Automatic identification of items requiring re-ordering;

* Routine material/component delivery control procedures;

* Provision of up-to-date physical and free stock figures, on order balance and allocated balance, per item;

* Identification of slow-moving, obsolete and over-stocked items;

* Full stock checking facilities;

* Provision of up-to-date stock valuation information.

FIGURE 1: SAVINGS AND BENEFITS POTENTIAL

BENEFIT	LEVEL OF SAVING EXPECTED
Increased Productivity	Highly engineered factory: 2-4% Small batch company: 5-10%
Reduction in level of work-in-progress	5-10%
Reduction in factory excess costs	5-10%
Reduction in factory-related clerical costs	10-40%
Reduction in shipping costs	10-50%
Improved morale of supervisors/managers	Not quantified
Improved delivery performance to customers Improved customer service	Not quantified, but should generate additional sales

REAL TIME SAVINGS AND BENEFITS

Obviously the level of savings and benefits which can be realized in a specific situation depend both on the effectiveness of control systems currently in use, and

also on the type of manufacturer who would use the system. A highly engineered company manufacturing a relatively stable range of products on a contract basis will gain different levels of savings and benefits to an organisation making small batch lots of a wide range of differing products.

However, in Figure 1 we outline the type and level of benefits, including quantifiable savings, which can accrue from the use of this type of real-time production control system.

ROI POTENTIAL

For a medium-sized manufacturer, having about 250 direct operators, the payback period for the PCI SYSTEM is often 1-1.5 years. As the installed cost of the system works out at approximately 75 cents per direct operator per working day for five years, it can be seen that a very attractive return on investment potential can be realised from the use of this type of installation.

CONCLUSIONS

Various techniques for the efficient control of manufacturing have been outlined above. Certain disciplines need to be introduced and all levels of staff fully trained if these control systems are to be successfully implemented. Key, up-to-the-minute information for action will result from the introduction of a real-time system; however, it is the use of this data which will determine whether or not the potential benefits from improved control systems are realised.

If manufacturers in developed countries are to remain competitive, or even survive, then modern methods of management control must be utilised in their factories. The PCI On-Line Production Control System provides the type of dynamic factory control system necessary in today's working environment to ensure that maximum productivity, minimum factory operating costs and a high level of delivery performance is achieved. We suggest that these three factors are essential to the successful operation of any manufacturing business today and that therefore this type of system should be seriously evaluated by any progressive company.

1023. SARSAT - SATELLITES, HPs, AND SEARCH AND RESCUE

Norman P. Livermore
Lt. Edward Blanchard
TSA, Inc.
Suite 325
1400 Lake Hearn Dr.
Atlanta GA 30319

Most general aviation aircraft and many boats carry some type of Emergency Locating Transmitter (ELT) which uses a special radio frequency for reporting emergency situations. The major problem with these transmitters is detectability because most distresses occur in isolated areas. Quick location and recovery time are critical to the survival of the victims of a crash or other distress. Studies show that only 20 percent of injured victims will survive if not rescued with 24 hours and only 50 percent of uninjured victims will survive if not located within 72 hours. Prior to SARSAT, an airport or harbor monitoring station, aircraft or ship in the distress area and listening on the proper frequencies had to receive the distress signal for rescue to be effected. SARSAT (Search and Rescue Satellite Aided Tracking) and the Soviet equivalent COSPAS has brought a space-age solution to this problem. With SARSAT, U.S. and Soviet satellites regularly pass over the earth in low, near polar orbits listening for distress signals and relaying them to Ground Stations throughout the world.

SARSAT/COSPAS is a cooperative effort spawned as an experiment involving the U.S., Canada, France and the Soviet Union using satellites to detect and locate these emergency signals. The project quickly expanded, bringing in the United Kingdom, Norway, Finland and Bulgaria. The most recent tally of SARSAT related rescues is 343; 154 of which are Marine, 177 of which are aircraft, and 12 of which are classified as terrestrial.

The United States involvement in SARSAT may be divided into two parts: A Spaceborne segment and a ground based segment.

The entire project is based on the ability of low earth orbiting satellites to receive distress radio signals and relay information about them to earth based rescue forces. There are two discrete components of the spaceborne segment of the project; the first is the spacecraft. The current environment consists of three Soviet Cosmos class satellites called COSPAS I, II, and III and two United States rescue packages called SARAT I and II which ride on specially equipped TIROS-N satellites. All five satellites are in a high inclination polar orbiting constellation at an altitude of approximately 450 miles and with an orbital period of about 100 minutes.

Each Satellite is equipped with radio receivers for 121.5 MHz and 406MHz radio signals. The SARSAT packages include an additional capability for receiving 243.0 MHz distress signals. This last frequency band is reserved for U.S. Military use. These distress signals are transmitted to ground stations in "real time" when the distress and the ground station have mutual visibility of the spacecraft. When the ground station cannot receive the signals, each spacecraft has the capability to store them and replay them upon command.

The second component of the project's space segment are the distress signal transmitters themselves. Called Emergency Locator Transmitters (ELT) or Emergency Position Indicating Radio Beacons (EPIRB), they are simply a self-contained, battery powered radio transmitter. These transmitters have an automatic trigger. ELTs are aviation type transmitters and are required to be carried on all civil aircraft. ELTs are designed to be automatically activated upon crash through the incorporation of "G" switch. EPIRBS are carried by the marine community and are designed to be activated automatically upon floating. A special type of ELT is also available which is called a personal ELT. This device is designed to be carried on backpacking and other outdoor activities in remote areas. These ELTs may be activated by manual means only.

406 MHz beacons differ from the 121.5MHz and 243.0 MHz beacons by providing unique identification and information on the type of distress and number of persons involved. This generation beacon is currently being developed. Tests of the beacon show highly favorable results.

The United States ground segment consists of a network of ground stations called Local User Terminals (LUTs) and the United States Mission Control Center (USMCC).

The LUTs are responsible for tracking, receiving and processing distress signals relayed from the satellites passing within site of the LUT antenna. An HP-1000 F-series computer is the heart of each LUT and is responsible for controlling all LUT processes. The LUT configuration consists of the following hardware and system software.

1. HP Configuration
   - HP-1000 F-series processor with 1 megabyte of memory.
   - Two 7925 MAC Disc drives.
   - One 7970 tape drive.
   - HP 12966A BACI cards, terminals and printers.
   - HP 12794B DS-1000 Modem interface card for MCC communications.
   - HP 12825 CPU to CPU direct connect DS-1000 cards.
   - RTE IVB operating system
   - DS-1000 IV network software for remote communications
2. Other Hardware
   - Datron antenna subsystem
   - AP-120 Array Processor

The software system within the LUT is written primarily using FORTRAN 4 with several special purpose modules written using HP1000 Assembler. Assembler use is restricted to those functions where timing requirements are critical ("Real Time" processing functions). The LUT software can be separated into four basic sub-systems: Pre- Pass processing software, "Real Time" processing software, Post-Pass processing software, and User Interface software. Pre-Pass software performs all operations necessary to ensure that the LUT system is ready to receive data from an upcoming Satellite pass. Within the Pre-Pass subsystem the following functions are performed: the antenna and receiver system is positioned, all data files required during "real-time" processing are verified present and properly initialized, all inter-pass processing software is initialized and ready for the satellite pass, and finally the Pre-Pass subsystem is responsible for initializing the array processor software and the signal processing hardware.

The Real Time processing software is responsible for collecting all data required for Post-Pass processing during the Real Time tracking of the spacecraft. This software collects the 121.5 MHz and 243 MHz data from the array processor, separates it into the different bands and places it in memory for future processing. The Real Time software is also responsible for collecting all 406MHz signals from 2.4K bit frame synchronizing hardware responsible for receiving and decoding the 406 MHz signals. Also collected during operation of this subsystem is range rate data used during Post-Pass processing for satellite orbital elements. The Real-Time subsystem is also responsible for controlling the antenna's position during the satellite pass. Post-Pass processing is the subsystem where the actual signal processing is performed. 406MHz signals are processed through a weighted least squares processing technique to obtain positional information (lat/long). 121MHz and 243 MHz signals are also processed into position information during Post Pass Processing. These positions are computed using a two dimensional FFT algorithm. Both raw signal data and doppler templates created by signal characteristics and spacecraft/LUT geometry are applied to the FFT logic. The positional data is placed in memory and transferred to the USMCC when the Post Pass processing cycle has been completed. The User Interface software is provided to permit the LUT operator to interact directly with the LUT processing software. A set of user friendly command oriented modules are included in this sub-system allowing the operator to modify controlling parameters, as well as permit the human operator to assist in the system decision making process.

A set of utility software is also present within each LUT to permit reprocessing of previous satellite passes as well as to provide data analysis and debugging capabilities.

The USMCC acts as a focal point for all North American SARSAT/ COSPAS related communications. The USMCC software is responsible for collecting all US LUT distress locations, sorting these locations by geographic region and transmitting the locations to the proper rescue coordination center worldwide. The USMCC is also responsible for processing all SARSAT stored data including stored distress data, spacecraft instrument housekeeping information (Telemetry) and management of all SARSAT Spacecraft orbitography information with regards to the worldwide ground station network.

The USMCC uses two HP-1000 F-series computers operating as co-processors. The actual configuration of the USMCC hardware and system software is as follows:

1. HP Configuration
   - Two HP-1000 F-series processors
   - Four HP-7925 MAC disc drives
   - Two 7970 tape drives
   - A mix of 26XX terminals
   - HP 12828A 8 channel MUX for six HP terminals
   - HP 12828A 8 channel MUX Tektronix terminal interfaces
   - HP 12966A BACI cards for terminal interfaces
   - 12821A HPIB used with extenders for IBM-PC, HP printer and HP 9872C Plotter
   - 12794B DS-1000 Modem interface board for each of the LUT modes.

2. Non HP Equipment
   - IBM PC used in the RCC connected using a Tekmar IEEE-488 interface board to the HPIB extender.
   - Tektronix 4115 used in the RCC via the 8 channel MUX.
   - Nu data T-16 Telex interface.

3. The USMCC computers both use the RTE-6/VM operating system and both have two megabytes of main memory. All communications with the LUTs is through HP's DS-1000 networking software. All incoming and outgoing data is kept in IMAGE data-bases and being installed are DSN/X.25 1000 and PMF/1000. The X.25 interface will replace the current synchronous communications interface while the PMF/1000 will connect the USMCC system to an IBM 4341 located in the US Air Force RCC.

The USMCC processing system SARIPS (Search and Rescue Information Processing System) resides on two HP1000 F-series Computer systems and is comprised of 52 separate software modules. The SARIPS software is written using FORTRAN77 with several fundamental routines written in HP1000 Assembler. Interprogram communication is achieved using RTE class input/output operations. A master scheduling routine SCHED provides a means of programatically assigning class buffers to each SARIPS module and initiating operation of the module (program scheduling). The DS1000 DEXEC utility is used to schedule those modules identified to execute on the processor which is not "controlling" system initialization. As each module is scheduled it sets up its own operating environment, responds to the Master Scheduler and examines its assigned class queue for incoming commands.

USMCC-LUT communication is accomplished by software which takes advantage of the HP DS1000 subsystem using Program-to-Program communication operations. As data is received from a LUT it is identified according to data type, LUT identifiers, spacecraft source and orbit number. Inputs are collected and sorted by this identifying information and processed as a collection. The "best" position is extracted for each location set using signal characteristics in a probabilistic module. Where possible the ambiguities of the LUT doppler processing are resolved by historic comparisons to other satellites and earlier orbits. After the "best" solution has been determined, the position is used to determine the rescue region of responsibility. Rescue regions are determined according to a predefined set of latitude/longitude pairs which define international boundaries as well as domestic rescue regions. Information pertaining to each distress signal processed is then formatted and transmitted to the responsible parties.

Stored SARSAT data is received at the USMCC in "raw" form from NOAA's data processing sub-system (NOAA-DPSS). This data is transmitted by a special communication interface called NASCOM. The data received is identified by spacecraft and data type, formatted into a usable form and passed to one of several processing modules. Data identified as incident (containing distress data) is passed on to be processed into location/frequency information using a method similar to that employed in the LUTs. This information is then collected again by satellite and orbit and passed to the geographical identification software to be merged into the "Real Time" incident data processing flow. Spacecraft housekeeping data, identified as telemetry data, is separated into

data related to the Search and Rescue Repeater (SRR) provided by Canada's DOC and the Search and Rescue Processor (SRP) provided by France. This separated data is passed to the appropriate processing modules where the digital data is converted into voltage, amperage and temperature information. These values are used to perform statistical analysis on the general health of the respective instruments. Reports containing this information are formulated and transmitted to the proper foreign agency.

The USMCC requires all data received and transmitted be archived for recall at a later date. The IMAGE 1000 system is used for this purpose. Two databases have been created for incoming and outgoing message traffic respectively. Several user oriented software modules are contained within SARIPS to permit USMCC personnel to obtain information on message traffic, extract and display a message according to any of several keys, or retransmit a message as requested by a user. These programs are menu driven using soft-key technology. A separate master menu program is available to provide online documentation on all other user software and well as provide one key execution of a selected program.

Graphical representation of spacecraft suborbital tracks according to time, and ELT/EPRIB activity in a given area are also included in the SARIPS software. Menu driven programs provide graphical information to USMCC personnel as requested. This information is dispatched to an HP Color Graphics terminal or an HP eight pen plotter according to user inputs. Another graphic based module is contained within the system which automatically displays ELT/EPIRB activity within the continental United States on a set of TEKTRONIX Color Graphics Terminals.

A final module is present in SARIPS. This module, called MONTR, provides automatic monitoring of all SARIPS software and adjusting priorities of modules which show above a given level of backlog in their class queues. The module also provides for a certain level of automatic error recovery and notification of any trouble to the USMCC personnel. The monitor software automatically checks the status of all communication interfaces by "looping back" through the distant end at periodic intervals. MONTR results and trouble reports are displayed on a screen format resident on an HP Color Graphics Terminal.

As the USMCCs chief role is data collection and distribution in a worldwide network, several different communication interfaces are contained within the system. These interfaces include: DS1000 based computer-to-computer communication used for interfacing the USMCC and all US LUTs as well as between USMCC processors, an HPIB communication with an IBM PC used for automatic record keeping within the Air Force RCC, an AUTODIN interface (Automatic Digital Information Network) used to relay information to the US Coast Guard RCCs and between the USMCC and the Canadian MCC, a TELEX interface for communications between the USMCC and all European and Soviet MCCs, and a special synchronous communication interface called NASCOM used for USMCC/NOAA-DPSS communication (relay of SARSAT Stored Data). These interfaces are described below:

A. IBM PC
   1. Specifications and hardware.
      - Tekmar IEEE 488 IBM PC interface.
      - HP interface bus (HPIB) with extenders.
      - Data Products DDU-1 short haul modems.
      - 9600 baud transfer rate.

2. Description.
The IBM PC is used by the Air Force RCC as file system and a means of keeping track of the current status of search and rescue events. Prior to the connection of the PC to the HP-1000 all SARSAT data was entered manually. This was a time consuming job. As a part of this design a program was written to receive data bound for the PC and hold it until a request to send was received from the PC. Hardware communication was established using a Tekmar IEEE-488 interface card connected to the HPIB bus. Periodically the PC sent the HP a data request. If the HP had no data it returned a no data flag. If the HP had data it sent the data via the HPIB bus.

B. AUTODIN.

1. Specifications and hardware.
   - HP 12966A BACI card.
   - Data Products DDU-1 short haul modem.
   - Western Union portable terminal controller (PTU).
   - 2400 baud RS-232.

2. Description.
Communication with Canada and U.S. Coast Guard is VIA the military AUTODIN system. AUTODIN is used because of it's low cost and availability to most Coast Guard RCCs. The HP portion of the interface uses a BACI card and the user group driver DVF00. DVF00 is used here and in TELEX because of it's very flexible nature. The other part is a custom Western Union portable terminal controller (PTC) Data is transferred between the BACI card and the PTC in 5000 word or less blocks. DVF00 is programmed to detect the program a line at a time. For transmission to AUTODIN the data is fixed record length.

C. TELEX

1. Specifications and hardware.
   - Nu Data T-16 Telex interface set.
   - HP 12966A Baci card.
   - 100 baud RS-232 between the BACI and the T-16.
   - 50 baud over the Telex network.
   - User group driver DVF00.

2. Description.
The Telex interface is by far the most troublesome one within the system. The Telex system was designed with very slow online terminals which had a high tolerance for errors induced by poor signals and bad connections. Automating this connection has been a challenge. The hardware level of interface is BACI card on the HP connected to a Nu Data T-16. DVF00 is again used as the driver. The T-16 handles the electrical interface, dialing and automatic answering of incoming calls. When a message is to be sent, a program called TELCM sends the T-16, via DVF00, a connect signal. The T-16 dials the network and sends the response back via the same route. If the response is a "go-ahead", TELCM sends the TELEX number to be dialed to

the T-16. The T-16 dials the number and returns the response to TECLM. TELCM checks this response and if it is correct, the message is sent one line at a time to the T-16. After the last line is sent, TELCM has the T-16 request confirmation of the transmission from the receiver. If the correct response is received, the transfer is complete. If not, the process is repeated until the message is successfully transmitted.

Incoming calls are answered by the T-16 which signals DVF00 that a message is inbound. When the T-16 gets a "go-ahead" from TELCM, via DVF00 it sends the message in one line at a time. When TELCM detects an end of transmission, an acknowledgement is sent to the sender via the T-16. The answered acknowledgement is validated. If the answer is proper the message is accepted; otherwise the message is rejected.

D. Tektronix 4115.

1. Specifications and hardware.
   - HP 12828A 8 channel MUX.
   - Data Products DDU-1 short haul modems.
   - 9600 Baud RS-232.
2. Description.
   The Tektronix 4115 is used by the Air Force RCC to give a graphical display of the "conus" U.S. and any rescue efforts in progress. A data-base of current ELT locations as well as other rescue situations is kept on the HP. When ever a new ELT is located by SARSAT it is added to the data-base and transmitted to the Tektronix terminal via a 9600 baud RS-232 Mux port. If the RCC controller enters any data at the Tektronix terminal it is transmitted back to the HP data-base via the MUX.

E. NOAA Communications.

1. Specifications and hardware.
   - HP 12618 Synchronous Communications Kit (SCK).
   - Cross Systems Nascom Interface Unit (NIU).
   - Racal-Milgo 9601 Synchronous Modem.
   - 9600 Baud.
   - Two custom privileged drivers developed by TSA Inc.

2. Description.
   All stored data from the SARSAT satellites as well as spacecraft tracking data is sent from NOAA to the USMCC using a NASA protocol known as NASCOM. The NIU performs error checking on the incoming data then sends it in 1200 bit blocks to the receive portion of the SCK. An application program checks the error bits then accepts or rejects the block of data. If the data is rejected a request to retransmit is sent back to NOAA through the Transmit portion of the SCK and NIU.

Search And Rescue Satellite Aided Tracking

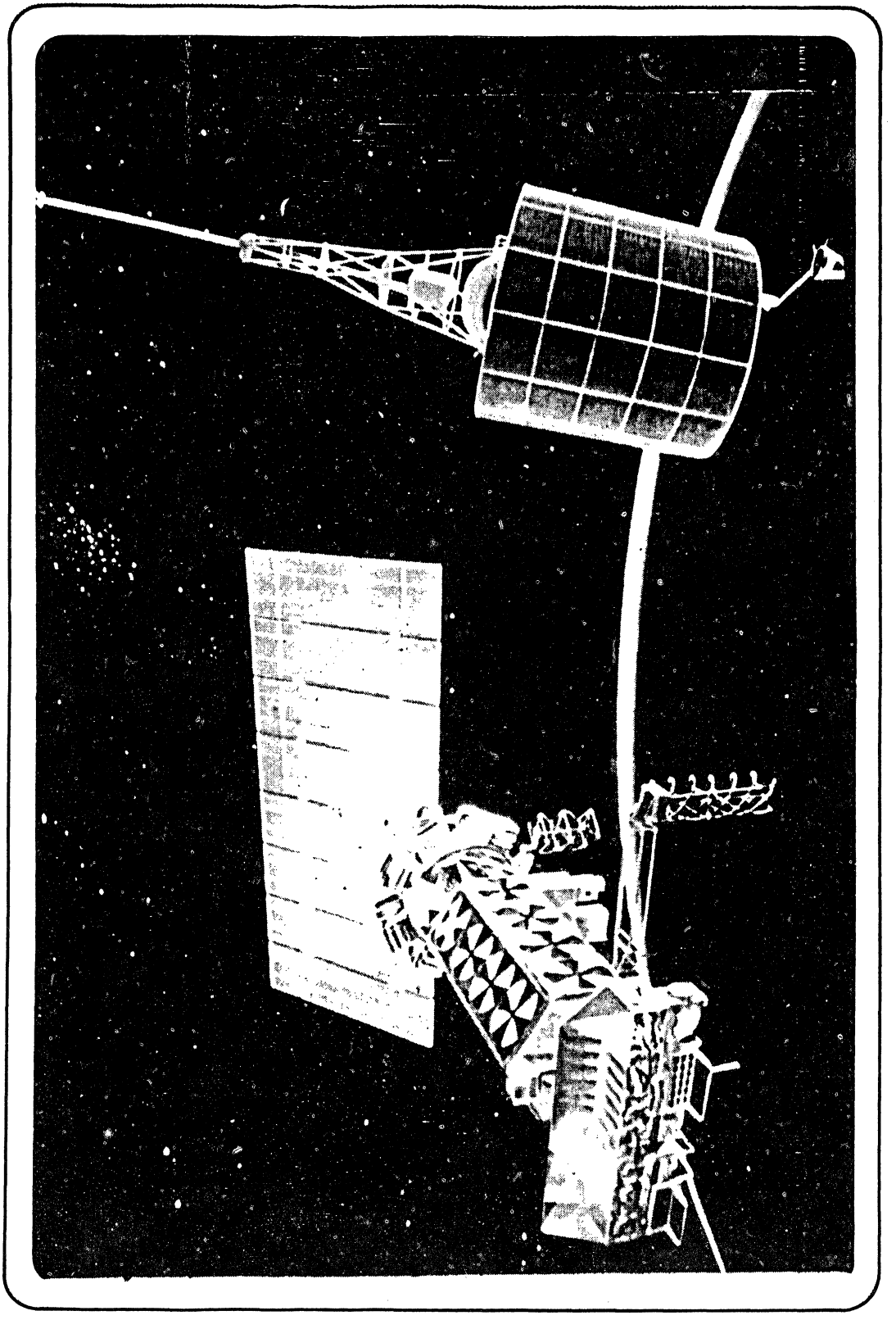
SARSAT

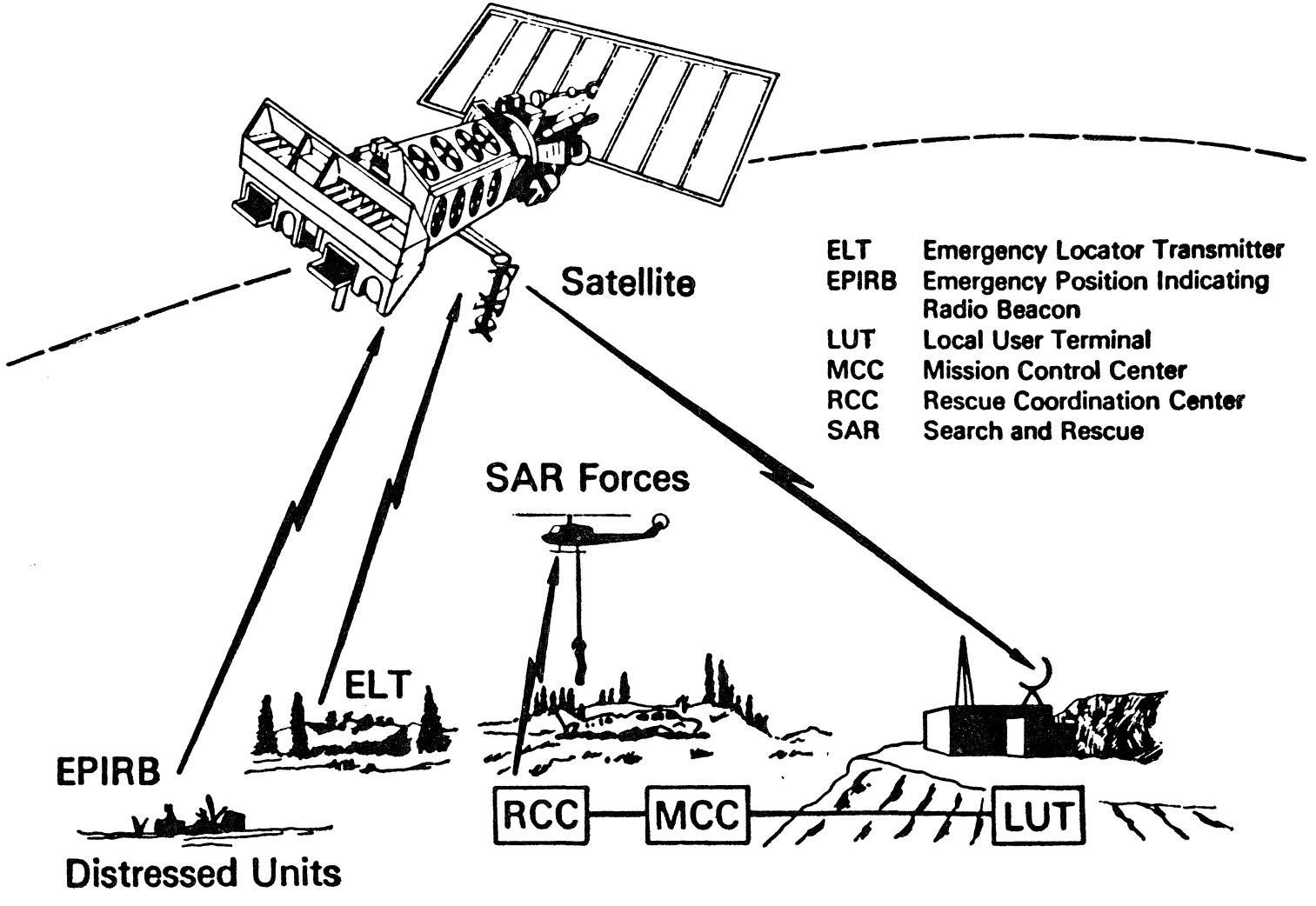
SARSAT OBJECTIVES

- REDUCED MANUAL EFFORT
- RAPID DISSEMINATION OF RESCUE INFORMATION
- IMPROVE LOCATION ACCURACY
- RAPID RESPONSE TO INFORMATION REQUESTS

SARSAT CAPABILITIES

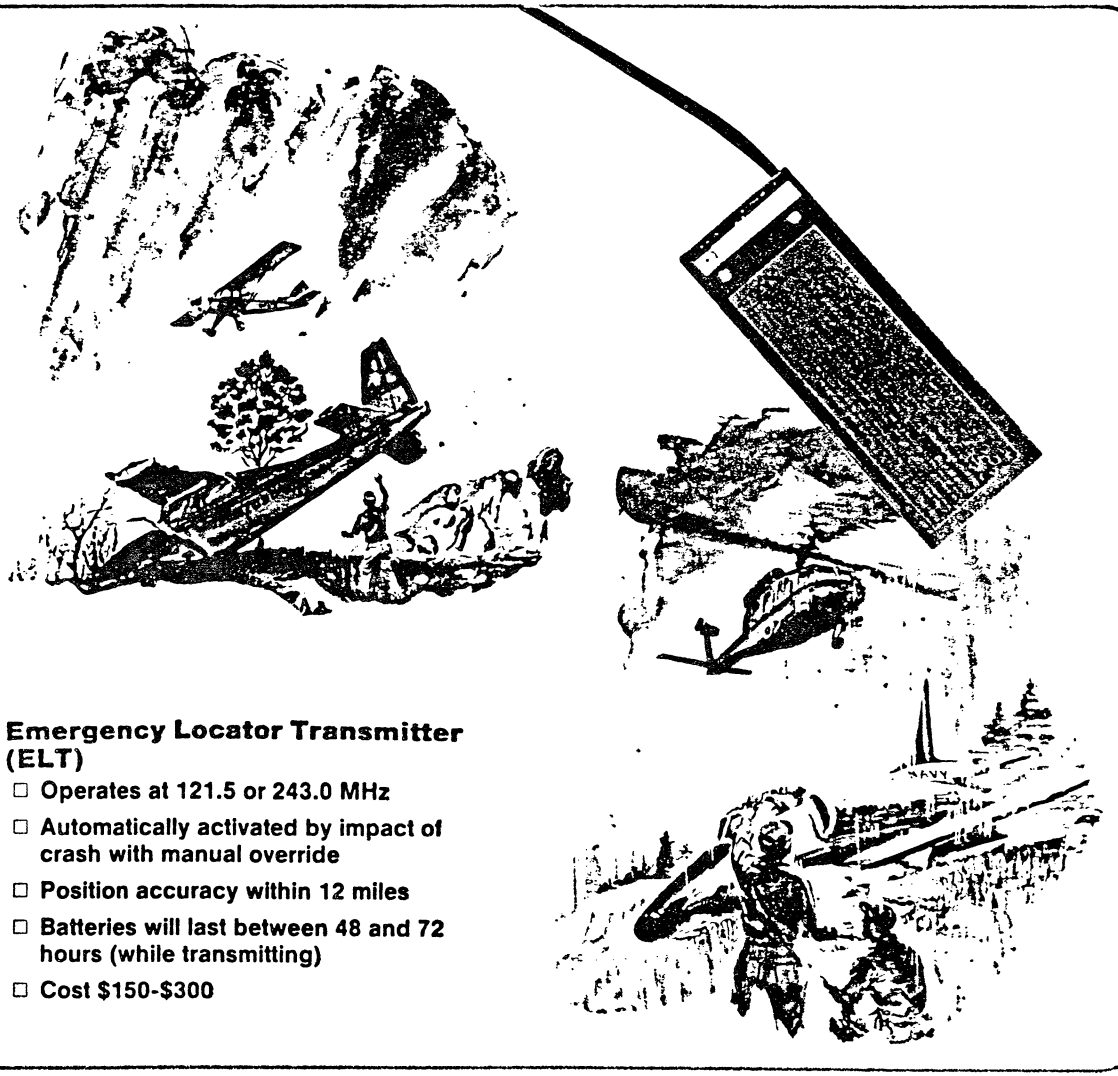
- AUTOMATIC DISTRIBUTION OF SATELLITE TRACKING DATA
- AUTOMATIC DISRTIBUTION OF SATELLITE STATUS INFORMATION
- AUTOMATIC MERGING OF MULTIPLE SOLUTIONS TO DETERMINE THE BEST LOCATION
- AUTOMATIC TRANSMISSION OF LOCATIONS TO THE PROPER RESCUE AGENCY
- INTERFACE WITH OTHER RELATED DATA SYSTEMS





SARSAT System Concept

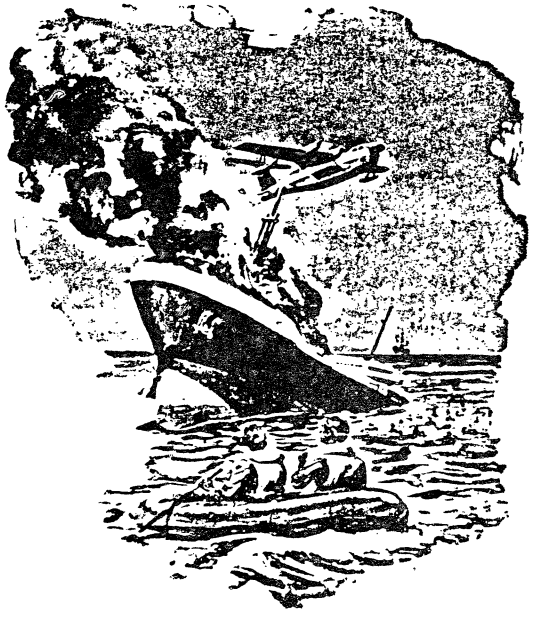
AIRCRAFT

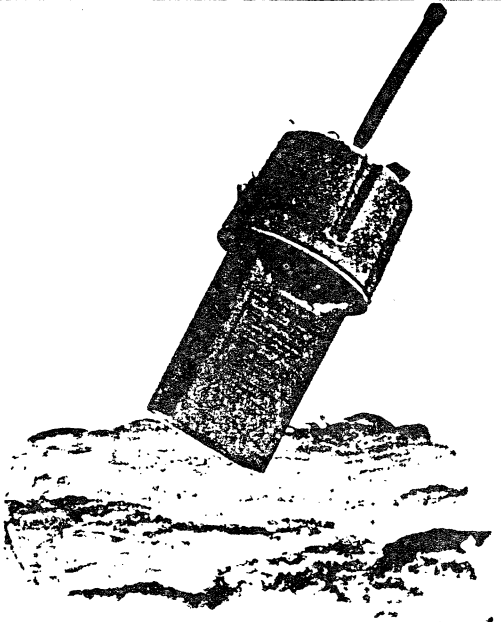


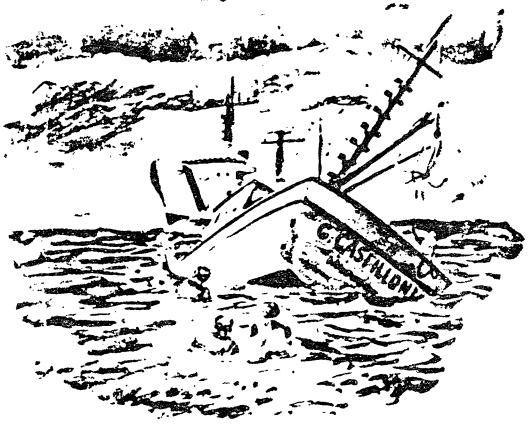
Emergency Locator Transmitter (ELT)

- ☐ Operates at 121.5 or 243.0 MHz
- ☐ Automatically activated by impact of crash with manual override
- ☐ Position accuracy within 12 miles
- ☐ Batteries will last between 48 and 72 hours (while transmitting)
- ☐ Cost $150-$300

MARINE



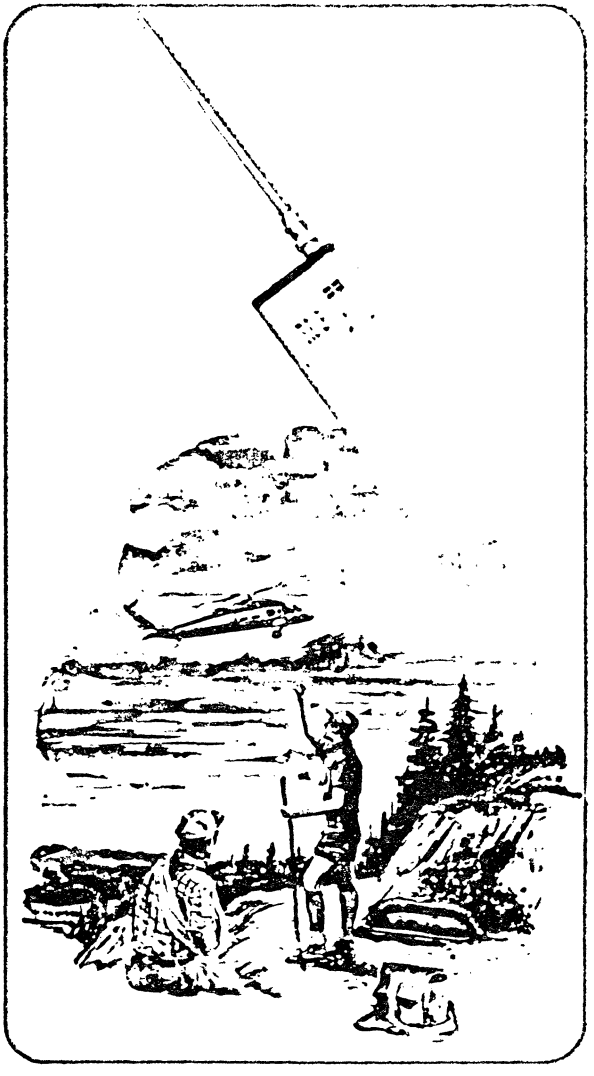


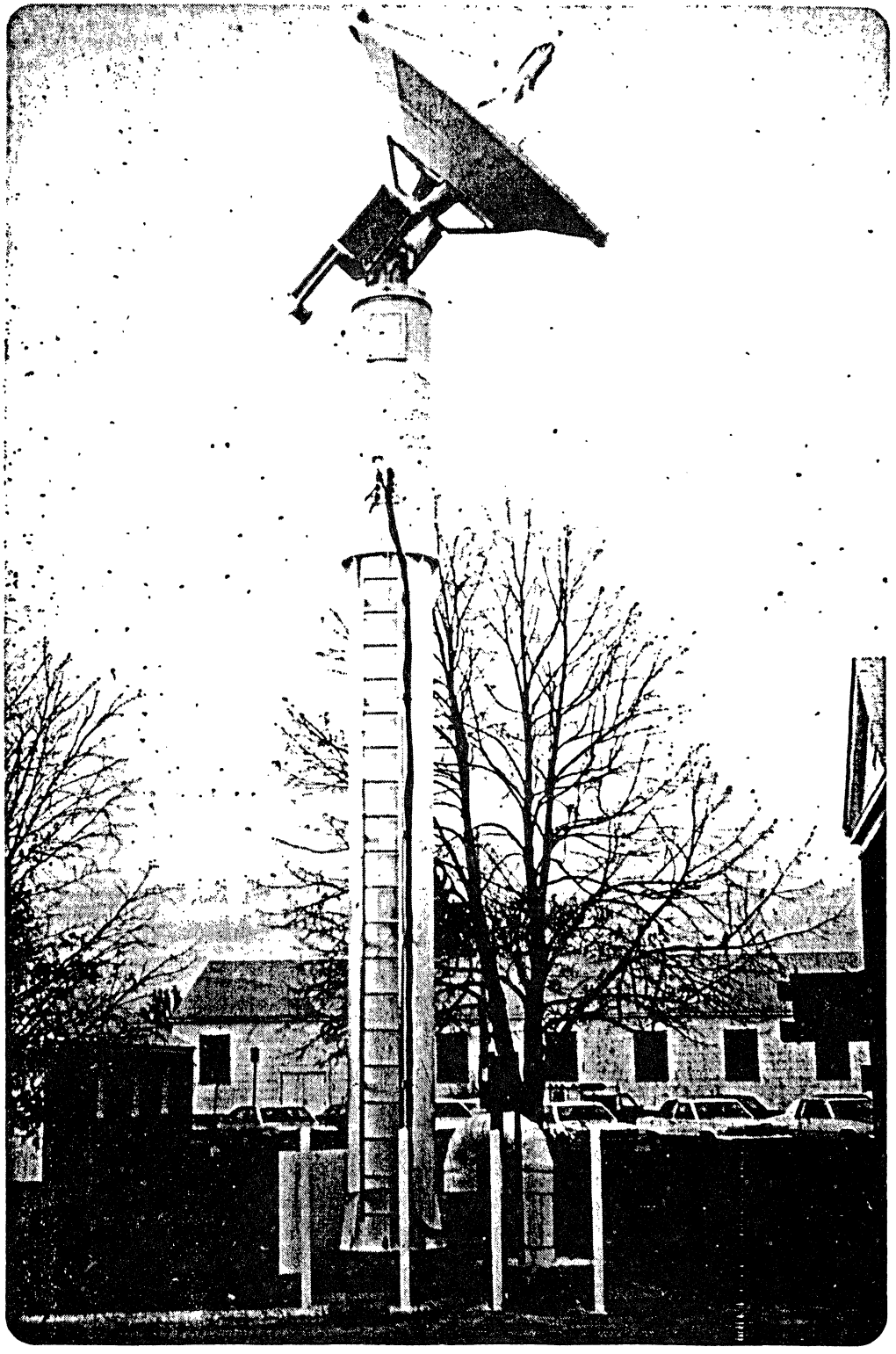


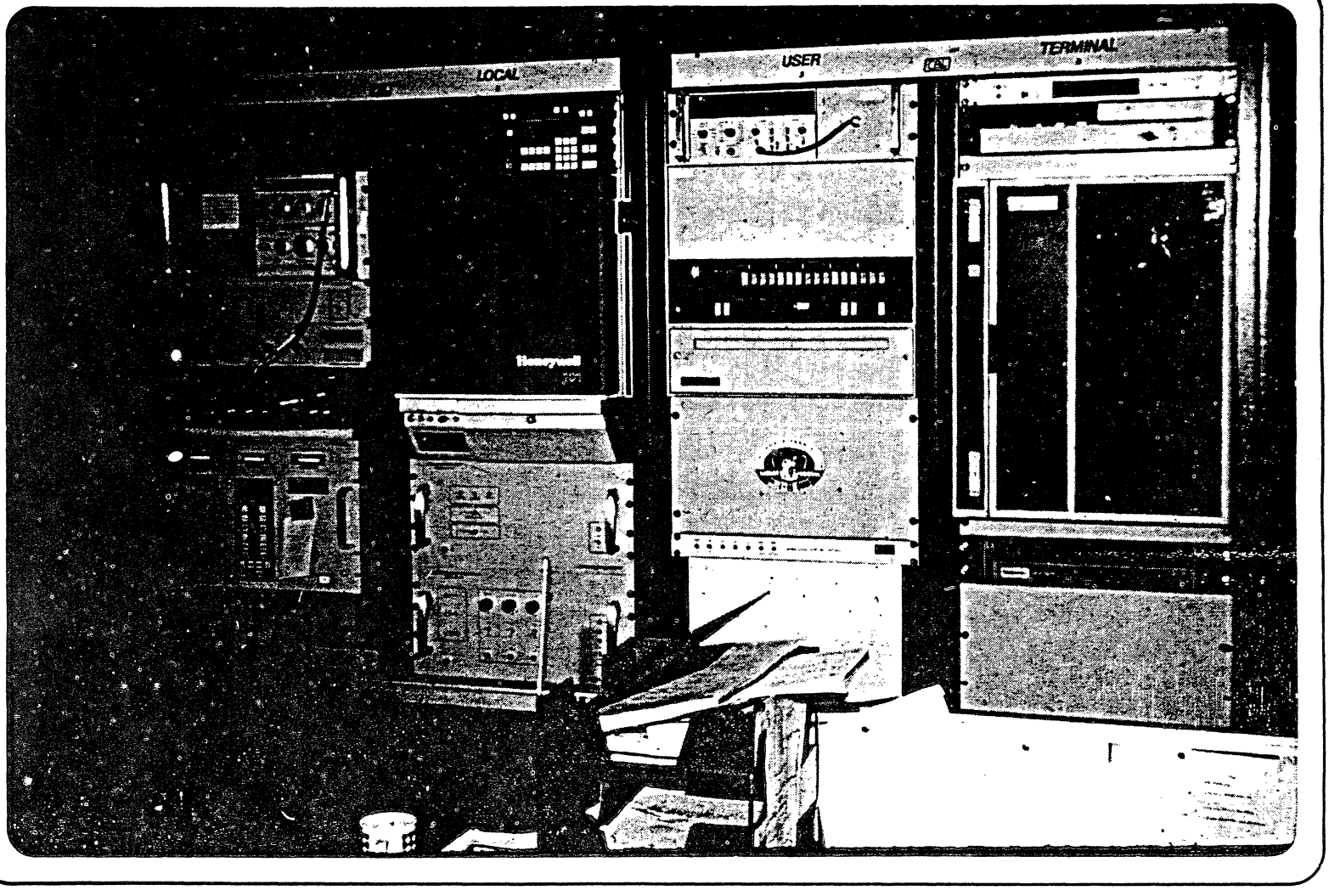
Emergency Position-Indicating Radio Beacons (EPIRB)

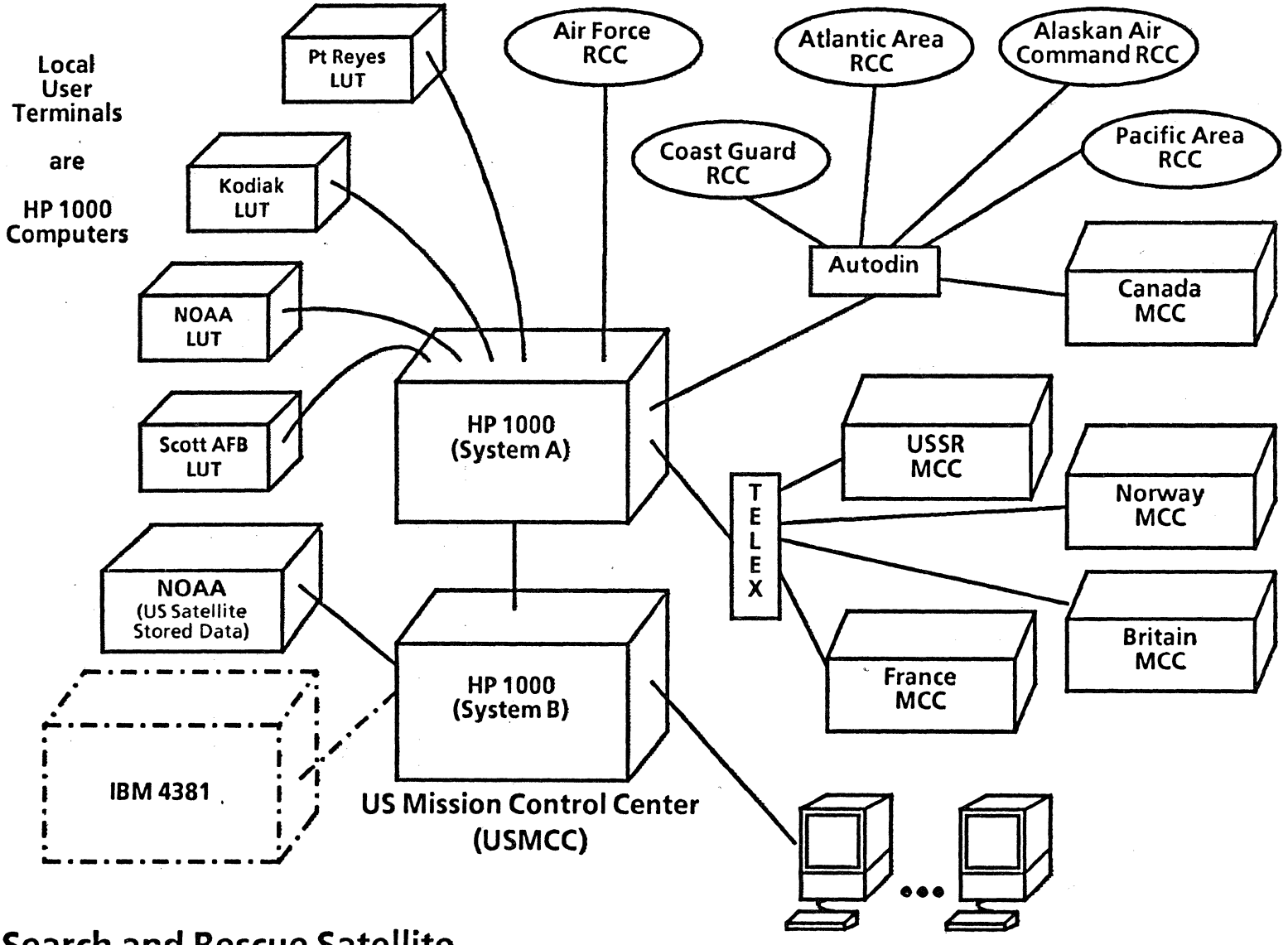
- ☐ Operates at 121.5 or 243.0 MHz
- ☐ Automatically activated by immersion in water with manual override
- ☐ Position accuracy within 12 miles
- ☐ Batteries will last between 48 and 72 72 hours (while transmitting)
- ☐ Cost $150-$300

REMOTE



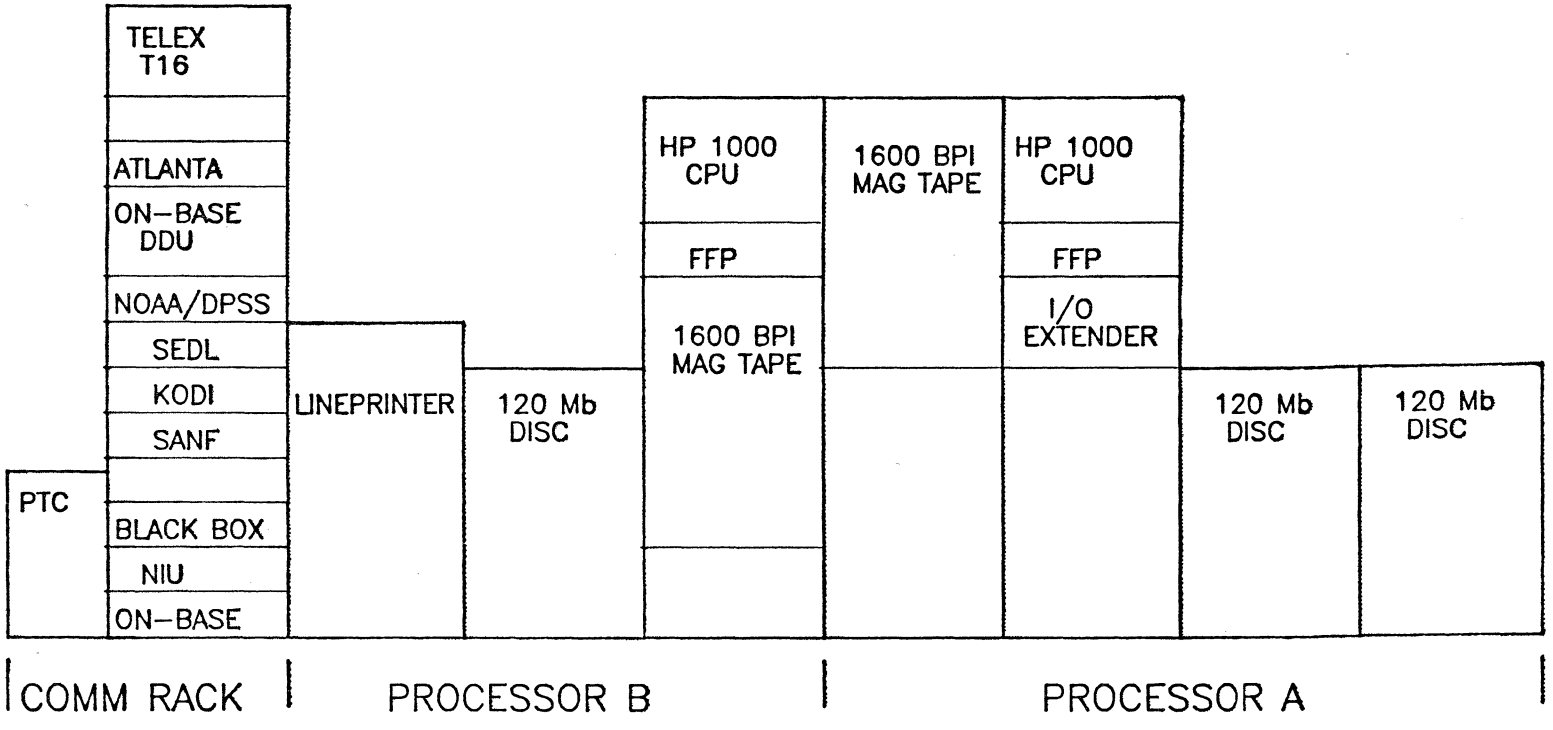



LOCAL
USER
TERMINAL

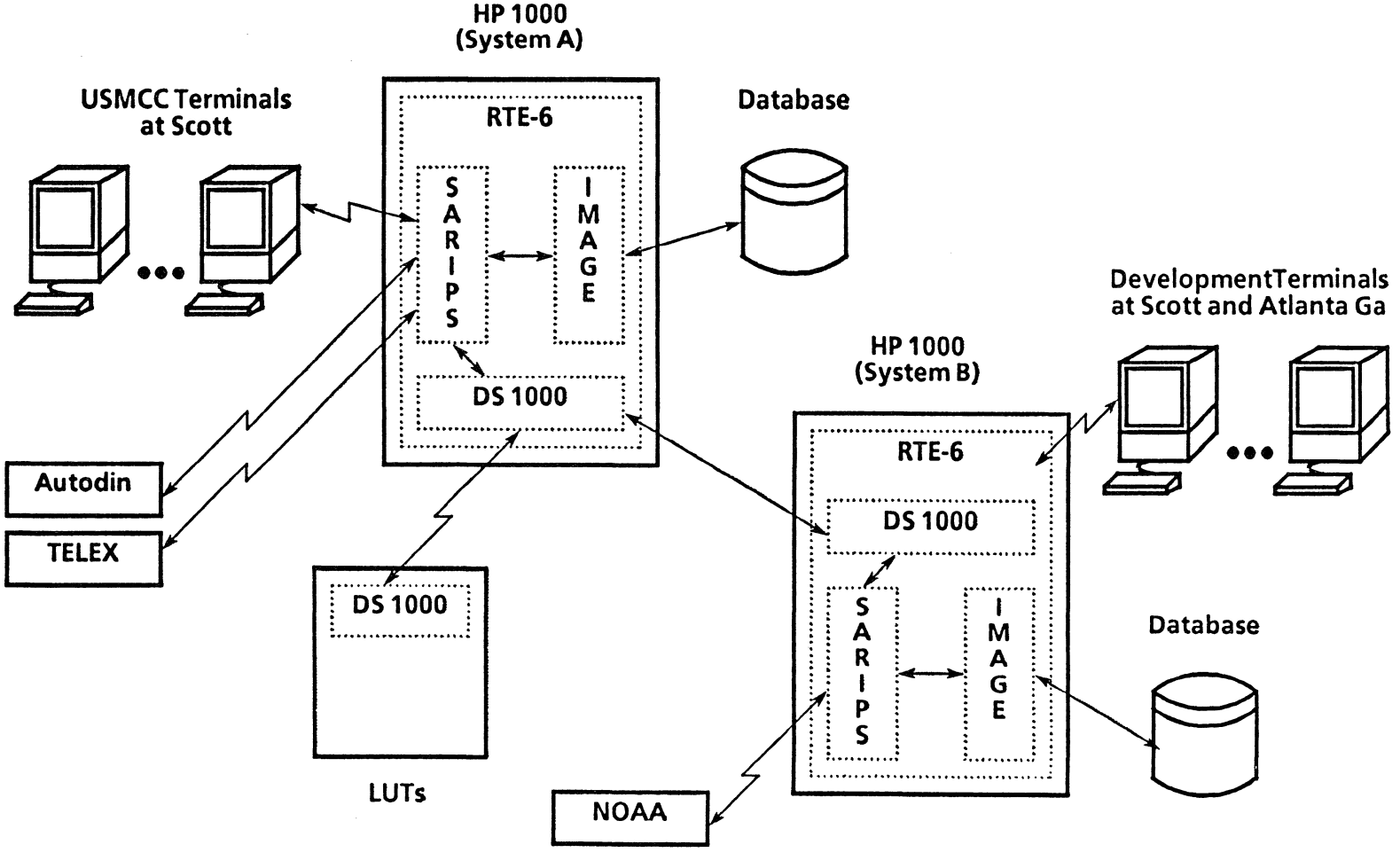


Search and Rescue Satellite-Aided Tracking System (SARSAT)

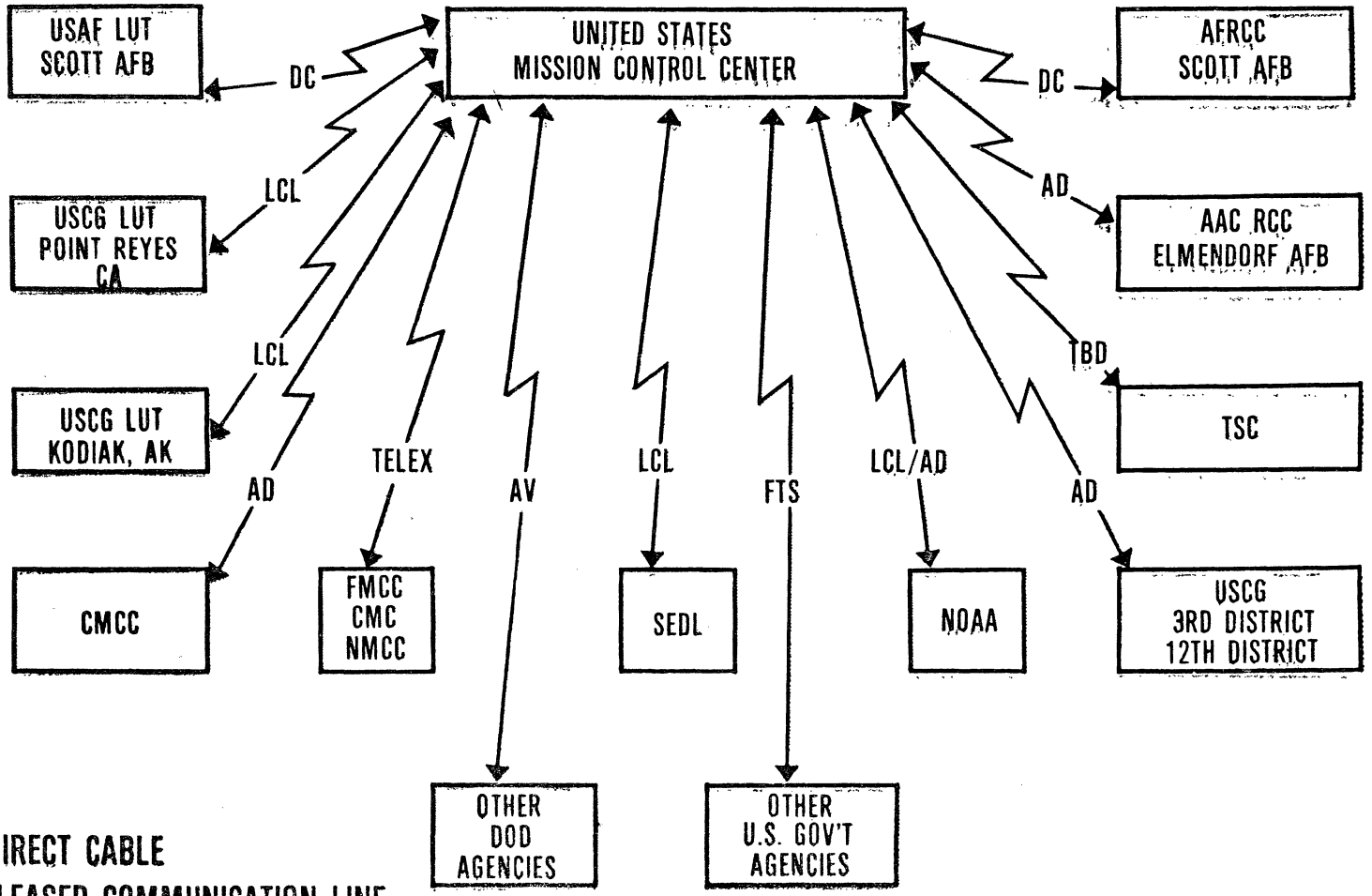
SARSAT HARDWARE CONFIGURATION



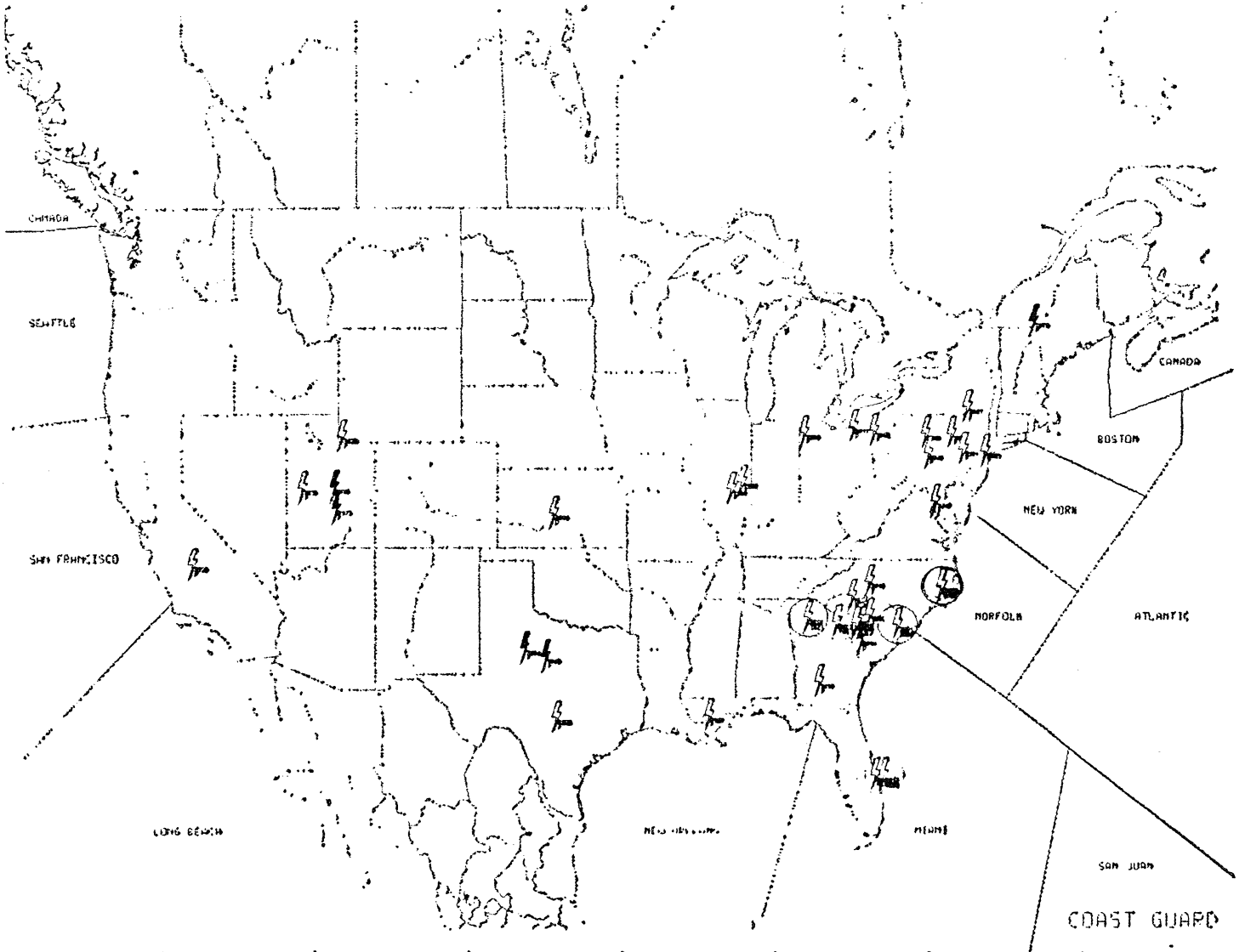
US Mission Control Center (USMCC)



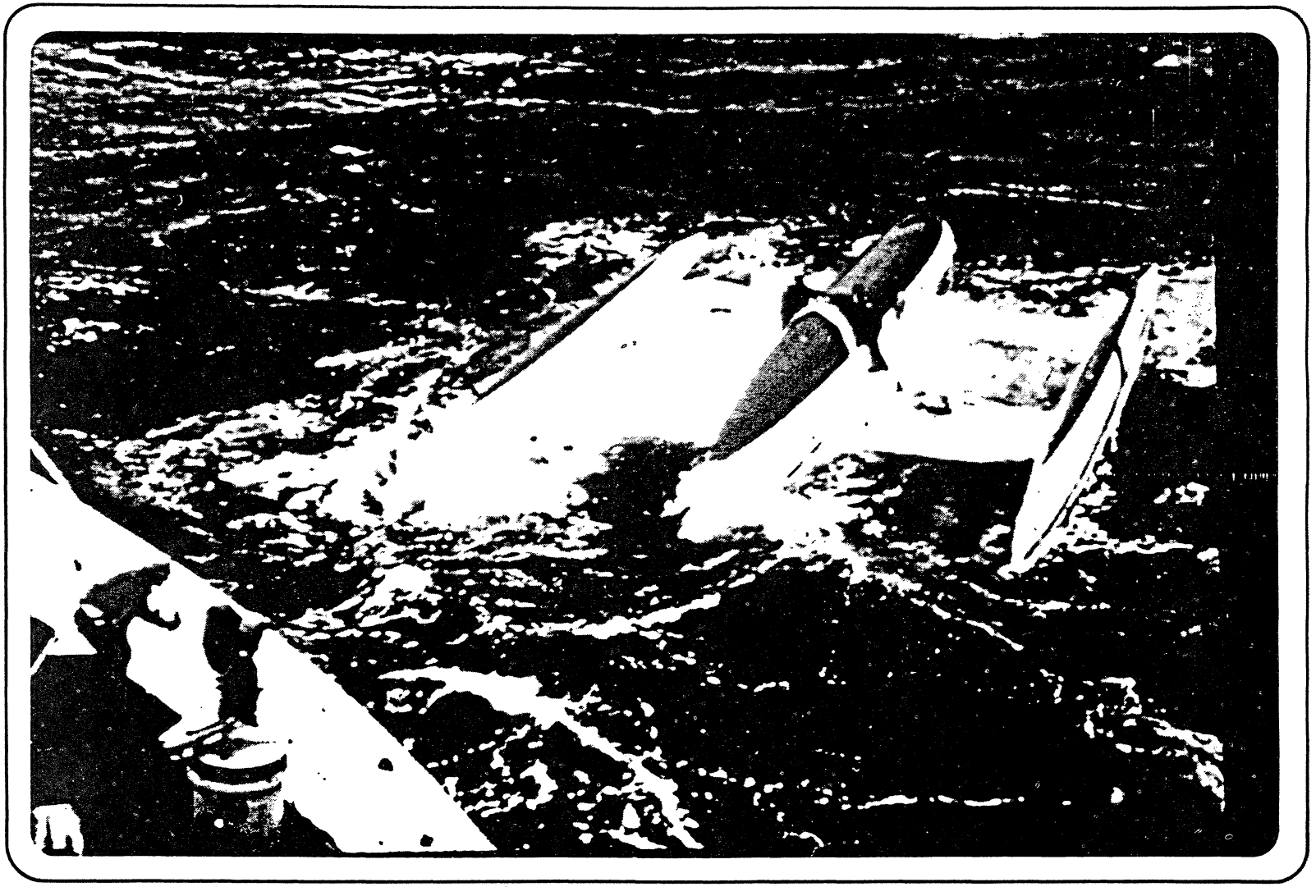
USMCC COMMUNICATION INTERFACES



DC - DIRECT CABLE
LCL - LEASED COMMUNICATION LINE
AD - AUTODIN
AV - AUTOVON
FTS - FEDERAL TELECOM SYSTEM



INPUT | MODIFY/ CLOSE | SORT/ REFRESH | CORRELATE | RECALL/ PURGE | REPORT/ LISTINGS | GRAPHICS | EXIT



1024. RTE-A COMMAND INTERPRETER FEATURES

Beth Clark
Hewlett-Packard Company
Data Systems Division
11000 Wolfe Road
Cupertino, CA 95014

FMGR didn't have

At Revision 2440, Hewlett-Packard significantly enhanced CI, the Command Interpreter program on the HP1000. CI's latest features include nestable command files, user-defined string variables, predefined system string variables, command file control structures, return string and return status variables.

These new features make CI more capable, powerful, flexible, and friendly than ever before. Procedures that were done manually before can now be automated and functions previously under program control can now be accomplished by command files. All of this adds up to increased engineering productivity.

The reader is assumed to be somewhat familiar with CI's commands and syntax, and the general concept of a command file.

Positional Variables

Positional variables (sometimes called positional parameters or positionals) are a simple way to pass values in to a CI command file. They are "positional" because the relationship between a variable and its value is determined by its position in a CI runstring or TR command. (The TR command may be explicit or implicit.) For example,

In CI runstring:

```
CI,commandfile.cmd,var1,var2,var3
```

In the TR command (explicit):

```
CI> tr,commandfile.cmd,var1,var2,var3
```

In an implied TR command:

```
CI> commandfile.cmd,var1,var2,var3
```

A maximum of nine variables can be specified. They can be separated by commas or spaces. The value of any omitted variable is set to null. Commas must be used to hold the places of omitted variables.

The length of each variable is restricted only by the maximum length of the total string containing the positional variables. It can be up to 256 characters long. Any character can appear in the string (although CI metacharacters must be quoted for correct parsing).

Positional variables are referenced within a command file by "$" followed by a number. "$1" refers to the first value in the runstring, "$2" to the second, and so on. If a position greater than nine is referenced, it is interpreted by CI as a positional variable less than nine followed by a number - for example "$23" is parsed as the value of $2 followed the character "3".

When CI finds a reference to a positional, the value from the runstring is substituted in the command line in place of the positional. If a null or illegal variable is referenced, a value cannot be substituted therefore the character string is left untouched. So "LI $0" would report the error "No such file $0".

The positional variables receive their values at invocation of CI or a command file. The values can be referenced as positionals only in the current CI or command file - if another copy of CI is run or another command file entered, new positional variables are created for that invocation. Unlike other types of CI variables, positionals cannot be altered within the command file.

As an example, consider an interactive command that copies a particular file from one specific place to another. Suppose that many files with similar names must be copied. (If only a few files were to be copied, the command stack could be used to edit the command several times.) A command file that specifies the static parts of the command can be created, and positional variables used to pass the differing part of the file descriptor.

Suppose that without using a command file the following commands accomplish the necessary copies:

```
co testfile4::datafiles /archive/testdata/test4>56
co testfile5::datafiles /archive/testdata/test5>56
co testfile6::datafiles /archive/testdata/test6>56
co testfile7::datafiles /archive/testdata/test7>56
```

The command file used to replace the above commands would contain:

```
co testfile$1::datafiles /archive/testdata/test$1>56
```

Specifying the needed file descriptor part for each invocation of the transfer file will cause the same file copies to be done:

```
tr,commandfile,4
tr,commandfile,5
tr,commandfile,6
tr,commandfile,7
```

We will see later how a variable can be used to create the differing part of the file name in this example.

In the next example, consider that many operations must be performed many times on a certain set of files. In this case, the file names are static but the operation needs to be manipulated. A command file can be created that contains the file names with the operation determined through one or more variables:

```
$1 file1
$1 file2
$1 file3
$1 file4
$1 file5
 . . .
```

This file can be used to purge, unpurge, or create files, but only on the current working directory. Adding another variable for the directory allows any copy of the file wherever it exists to be accessed:

```
$1 $2file1
$1 $2file2
$1 $2file3
$1 $2file4
$1 $2file5
 . . .
```

As it exists, the command file can be used for commands that require only one parameter - the target file name. But most file manipulation commands require at least two file parameters - a source and a target. Adding a third variable increases the available command set to include rename, copy, move, and others. If the source file name should reappear in the target name, file masking can be used.

```
$1 $2file1 $3
$1 $2file2 $3
$1 $2file3 $3
$1 $2file4 $3
$1 $2file5 $3
 . . .
```

In this file, $1 represents the desired file operation, $2 represents the source file path, and $3 represents the target file descriptor.

You can use this file in many ways:

To copy all the files from /john/local/ to /global/:

```
CI> filer co /john/local/ /global/@
```

To purge all files on the current working directory:

```
CI> filer pu
```

To display information about these files on the current directory:

```
CI> filer dl,,!
```

User-Defined Variables

User-defined variables are similar to positional variables but much more flexible. Like positionals, the value of a user-defined variable is substituted by CI when a variable reference is made. However, the value of user-defined variables is not related to the CI or TR runstring. Also, the value can be altered at any time within a command file or interactively. The names of user-defined variables are not fixed (like "$1", "$2", etc.) but are user-specifiable.

The user-defined variable name can be up to 16 characters long. It must begin with a letter or underscore and may contain letters, digits, or underscores. When parsing a command line, CI determines the termination of a variable by encountering a character that is not legal in a variable name, usually space, comma, or slash. The variable value may be up to 80 characters in length. It may contain any character.

A user-defined variable is created by the SET command. Both the name and the value are specified in the command. If the variable already exists, the SET command assigns it a new value. The UNSET command deletes a variable.

A user-defined variable is referenced in the same manner as a positional variable - "$<varname>". CI substitutes the variable's value in a command line when it finds a reference to the variable. Unlike positionals, user-defined variables are kept globally by CI. A variable set in a command file can be referenced interactively or in another command file and variables set interactively can be referenced in command files, no matter how deeply nested. However, CI "knows about" its variables only. A copy of CI run from CI will not substitute for variables defined in the father CI, only those defined in the son CI.

CI only performs variable substitution for one level. If the value of a variable contains a string that references another variable, the string will NOT be interpreted as a variable and NO substitution will be made. This is easily demonstrated by:

```
CI> return 1 2 3 4 5
CI> set e = '$return1 $return2 $return3 $return4 $return5'
CI> echo $e
$return1 $return2 $return3 $return4 $return5
```

The values of the return variables are not displayed because CI substituted at one level, for '$e', and did not substitute again for the value of $e.

A common application of user-defined variables is in file pathnames. For example, the pathname "/sysgens/rev2440" could be assigned to "gen" and referenced as:

```
CI> set gen = /sysgens/rev2440
CI> rtagn $gen/answer $gen/system $gen/snap $gen/list
```

Note that the terminating slash cannot be part of the variable's value because CI needs the slash to recognize the end of the variable's name.

Predefined Variables

There is a set of variables that CI defines and initializes automatically. These variables are called the "predefined variables". The values of the variables will appear when a SET command without parameters is entered (along with the values of other types of variables). The variables' values can be altered by the SET command, but the variables cannot be deleted by the UNSET command.

The predefined variables can be grouped into three catagories. One set of variables defines what actions CI will take in certain circumstances: $AUTO_LOGOFF, $LOG, $PROMPT, $RU_FIRST, and $SAVE_STACK. A second set can be used to determine the current environment in which CI is running: $OPSY, $SESSION, and $WD. The last set contains variables used by commands and programs to return status information upon completion: $RETURN1 through $RETURN5 and $RETURN_S.

Note that CI automatically updates the value of some of the predefined variables. In the case of the current session number variable ($SESSION), it is impossible to actually change the variable's value to be different than reality. At the completion of processing the SET command, $SESSION is reset to the current true value. The current working directory variable ($WD) is only updated after each WD command, so its value may be temporarily altered between WD commands. Its value may be altered interactively by the SET command or programmatically by the FMPSETWORKINGDIR call.

$RU_FIRST

When CI processes an input command, it first attempts to recognize the command itself. Examples of internal CI commands are WD, HELP, MO, and CL. Next it attempts to treat the command as an RTE intrinsic, such as BR, PS, and OF. If the command is neither a CI or RTE command, than CI assumes the command is referencing either a command file or program file to be executed. The $RU_FIRST variable comes into play at this point. If $RU_FIRST is true, RU is inserted into the command line, if false, TR is inserted. The normal mechanisms for program schedule and command file access are then used.

An application of $SESSION and $PROMPT

Being able to control the prompt that CI issues can be quite helpful. One interesting usage of the $SESSION and $PROMPT variables is to set the current prompt to include the user's current session number. For example:

```
CI> set prompt = 'ci'$session>
ci24>
```

(For an explanation of all the predefined variables, see the RTE-A User's Manual [HP part 92077-90002]).

Completion Status Information

CI supports two type of return variables: five single integer parameters and a string. Testing the single variables in CI IF statements is a straightforward way of determining the success or failure of an operation. Based on status, the command file may terminate, continue, issue an error message, or take whatever action is appropriate.

Return variables

Most CI commands and HP-supplied programs return status variables. The PRTN system subroutine can be used by any program to return up to five single integers for CI (or any scheduling program) to access. Also, the CI RETURN command will accept 1 through 5 return variables.

The return variable usage of selected HP commands and programs is supplied here:

Function	$RETURN1	Others
DL	0 if no error or -<error code>	2-5 not used
TM	0 if no error or -1 if error	2-5 = 0
WD	0 if no error or FMP error if error in wd specification (Note: $RETURN1 is not set for errors in command file portion of WD command)	2-5 not used
TR	determined by parameters, set to 0 if not specified	
IS	status of compare	2-5 = 0
LINK	if no error, 1-3 = program name, 4 not used, 5 " " if error, 1=numeric error, 2-4 not used, 5=0	
MACRO	number of errors	2-5 = 0
CI	Depends on last command executed: if no error, left intact from previous values if error, FMP error or -1, 2-5 = 0	
EDIT	not used	not used
RTAGN	number of errors	2-5 = 0
SAM	0 if no problems -1 if problems	2 = total SAM in words 3 = total free/problem words 4 = no. of free/problem blocks 5 = largest free/problem block in words

Return string

The return string can be used to pass back information that doesn't easily conform to five integer parameters. The return string can be a maximum of 80 characters long and may contain special characters and meta characters. A command file may access the $RETURN_S variable just like any other variable. A program that wishes to return a string for use by a command file may issue an EXEC 14 call with second parameter equal to two. The CI RETURN command can be used to return a string from within a command file.

Control Structures

CI offers two control structures in command files: IF-THEN-ELSE-FI and WHILE-DO-DONE. These structures can manipulate command file execution flow through decision branching and iteration. CI's control structures are both conceptually and syntactically similar to constructs found in higher-level programming languages.

Either control structure may be nested within itself or within the other type of structure. There is no arbitrary restriction on depth of nesting, but when CI's internal buffers for control structures become full, CI abruptly terminates execution of the current command file.

Decision Branching

CI's IF structure simply executes a command and branches depending on the outcome of the command - if success is indicated, the THEN branch is executed, if failure is indicated, the ELSE branch is executed. The command may take many forms - a CI command, a series of CI commands, execution of a program - any activity that provides a return status of 0 (success) or non-zero (failure). CI recognizes the end of the command(s) when it finds the THEN clause. If multiple activities are specified, the return status of the last is used.

The THEN clause can be a single command or set of commands. The ELSE clause is optional and may also contain one or more commands. An IF structure must be terminated by a FI command.

Both the IF and THEN can also be followed by no commands. A null IF is interpreted as true. A null THEN is simply a placeholder for an ELSE clause.

Using the HP-supplied IS program expands the types of conditions that can be easily tested in IF structures. IS compares two strings for equality or all five types of inequality and returns success or failure.

The following command file segment demonstrates a simple command in an IF command. The command file runs a program called PROBE to check the status of the DS link specified in the variable $node and transfers to a (nested) command file that copies files over the link if the link is operational. If the link is not operational, the UP_DS program is run to attempt to restore the link.

```
if probe $node
     then tr ds_copy $node
     else  echo ' Attempting to restore DS line'
           if up_ds $node
                then tr ds_copy $node
                else  echo ' DS line down - no copying'
                fi
     fi
```

The following command file, that could be called crushall.cmd, can be used to run a utility called CRUSH with two different runstrings. The concept demonstrated in this example can be generalized to create a command file to interface with any program, varying the runstring as specified in the positional parameters:

```
if is $1 = e
     then crush @.@.e
     else  if is $1 = s
          then crush @.@.s
          else  echo,'Usage: crushall,e/s'
                echo,'  e crushes all files everywhere'
                echo,'  s crushes files in this directory tree'
          fi
     fi
```

Iteration

Another CI control structure, WHILE, provides the capability to repeat one or more commands as long as a condition remains true. Like IF-THEN-ELSE-FI, the WHILE structure is comprised of several parts: a list of commands in the WHILE clause, a list of commands in the DO clause, and a terminating DONE command. In most usages, the DO portion will alter the WHILE condition so the iteration will eventually terminate. The WHILE condition is evaluated before the DO command(s) is initially executed, and before each re-execution.

The IS program is very useful in WHILE structures as well as IFs. It can be used to create comparison conditions in addition to command status conditions. (See IF discussion and RTE-A User's Manual for more information about IS.)

One of the most common applications of the WHILE structure involves numeric iteration - that is, looping based on incrementing or decrementing a counter to a certain limit. While CI currently does not offer an intrinsic calculation facility, a simple calculator is easy to create:

```
ftn7x,q,s
      program calc

c  Calculator program to perform simple arithmetic.

c  runstring:  ru calc operand1 operator operand2

c  results:  RETURN1 = status (0 = good, -1 = bad)
```

```
c            RETURN2 = resulting value of operation


      integer params(5),operand1,operatori,operand2,result
      character operator*2
      equivalence (params(2),result),(operator,operatori)

c Get parameters from runstring.

      call rmpar(params)
      operand1 = params(1)
      operatori = params(2)
      operand2 = params(3)

c  Initialize return parameters.

      params(1) = 0
      params(2) = 0
      params(3) = 0

c  Check for operator.  User may need help.

      if (operatori .eq. 0) then
         write(1,'("Usage: ru calc operand1 operator operand2")')
         params(1) = -1
         goto 10
      endif

c  Check the operator and perform appropriate arithmetic.

      if (operator(1:1) .eq. '+') then
         result = operand1 + operand2

      elseif (operator(1:1) .eq. '-') then
         result = operand1 - operand2

      elseif (operator(1:1) .eq. '*') then
         result = operand1 * operand2

      elseif (operator(1:1) .eq. '/') then
         result = operand1 / operand2

      else
         write(1,'("operator must be +,-,*,or /")')
         params(1) = -1

      endif

c  Return status in params(1) and result in params(2).

10    call prtn(params)
      end
```

This calculation program can be customized to provide additional capabilities appropriate for particular applications.

The following command file utilizes both the IS and CALC programs. The variable $count is initialized to 0 and incremented from 1 to 12. The variable is used for two functions - as the loop control and as part of the file descriptor. Note that during development of the command file, an error check could be included after the CALC reference. For a simple increment operation the check is unnecessary during normal operation.

The current value of the predefined variable $LOG is saved and restored by the command file. The temporary variable used is deleted by the UNSET command.

```
*
*  Compile and link programs TEST1 through TEST12.
*
set oldlog = $LOG
set LOG = off
*
   set count = 0
*
   while is $count lt 12 -i
   do
      * Increment counter.
      *
      calc $count + 1
      set count = $RETURN2
      *
      * Compile file and link if no errors.
      *
     if ftn7x test$count.ftn 0 -
         then link test$count.rel
         else echo <bells>
              echo $RETURN1' errors compiling TEST'$count
              echo ' '
      fi
*
   done
*
set log = $oldlog
unset oldlog
```

FMGR and CI comparison

Earlier versions of RTE contained a command interpreter called FMGR (File ManaGeR). It provided a set of commands that included support of command files. The capabilities offered by CI and FMGR are compared in the table below:

Feature/Capability	CI	FMGR
define variable values	set	SE
show variable values	echo	DP
return variables (status)	5 integers	5 integers
return string	80 chars	no
conditional branching	IF	IF
unconditional branching	no	crude (true IF)
structured IF-THEN-ELSE	yes	no
structured WHILE-DO	yes	no
runstring variables	9, total 256 chars	5, 6 chars each
user-controlled variables		
mnemonic names	1-16 chars	no
length varies	1-80 chars	2 or 6 chars
variable removal	unset	n/a
friendly syntax	yes	no
implicit substring	no	limited, via P & G
implicit concatenate	yes	no
implicit calculator	no	CA
implicit prompt/response	no	DP/TR
pause	no	yes
error action control	$LOG only	severity levels
return to specified level	no	yes
comments	yes	yes

Note that many of the implicit capabilities lacking in CI can be replicated via a program (a simple calculator program appears in this document). CI's return string function could be used to create a prompt/response program.

A possible application of the two sets of features is demonstrated here by implementing the same function in a FMGR command file and a CI command file. The function receives the name of a PASCAL source file and two options. If the source file is found (on the current working directory or CRN), it is compiled. If the compile is successful the program is linked. If the link is successful, the program is executed and its return status displayed. The input options are "s" to print the error list file if compile errors occur, and "k" to keep the relocatable file.

First, the FMGR version of the file:

```
:*
:*  usage:  PASLK <file name> [s] [k]
:*                s - redirect output to LU6
:*                k - do NOT purge the relocatable file
:*
:*  Purpose: This command file schedules the PASCAL compiler on the file
```

```
:*          passed as the first parameter.  If the file compiles without
:*          errors LINK is scheduled with the debug option and the
:*          resulting program is run once.  If there are compiler errors
:*          the list file is listed to the terminal or if the s option is
:*          used, to the printer (via PRINT).  The K option can be used
:*          to save the relocatable file created by MACRO.
:*
:*  Note:   The file name must be 6 characters or less.  The FMGR
:*          special character naming convention (&,%,etc.) is used.
:*
:IF,-36P,EQ,3,2
:DP,Usage: PASLK <file name> [s] [k]
::
:RU,DL,1G,,0
:IF,1P,EQ,0,3
:DP,1P,2P
:DP,No such file:,1G
::
:DP,Compiler scheduled.
:CA,6,1G
:CA,-15:P,-15P,AND,377B,OR,22400B
:RU,PASCAL,1G,'PERR,6G
:IF,1P,EQ,0,5
:IF,2G,NE,S,2
:RU,PRINT,'PERR
::
:LI,'PERR
::
:PU,'PERR
:DP,Link scheduled.
:RU,LINK,6G,$PLIB,+DE
:IF,3G,NE,K,2
:DP,Relocatable saved in file:,6G
:IF,1,EQ,1,1
:PU,6G
:DP,
:DP,Your program scheduled.
:DP,
:RU,10G
:DP,1P,2P,3P,4P,5P
::
```

Now the CI version of the command file:

```
*
*  Usage:   PASLK <file name prefix> [s] [k]
*                 s - redirect output to LU6
*                 k - do NOT purge the relocatable file
*
*  Purpose: This command file schedules the PASCAL compiler on the file
*           passed as the first parameter.  If the file compiles without
*           errors LINK is scheduled with the debug option and the
```

```
*           resulting program is run once.  If there are compiler errors
*           the list file is listed to the terminal or if the s option is
*           used, to the printer (via PRINT).  The K option can be used
*           to save the relocatable file created by MACRO.
*
*  Note:    The file name will have .PAS added to it.
*
if is $1 = ''
 then echo 'Usage: PASLK <file name prefix> [s] [k]'
 else echo ''
  if dl $1.pas,,0
   then
    echo 'Compiler scheduled.'
    if pascal $1.pas paserror.lst $1.rel
     then echo 'Link scheduled.'
          pu paserror.lst
          link $1.rel pascal.lib prg$SESSION.run +de
          if is $3 = k
           then echo 'Relocatable saved in file: '$1.rel
                else pu $1.rel
          fi
          echo ''
          echo ' Your Program scheduled.'
          echo ''
          ru prg$SESSION
          echo $return1 $return2 $return3 $return4 $return5
          echo $return s
     else
      if is $2 = s
       then print paserror.lst
       else li paserror.lst
      fi
     fi
    else echo 'No such file: '$1'.PAS'
   fi
fi
*
```

Conclusion

A comparison of CI and FMGR capabilities shows that each command processor offers a cohesive set of functions. Each contains some functions that the other lacks, and some functions are common between them. CI's superior control structures and modern syntax make it the processor of choice. Most of the functions that CI currently lacks can be provided by external programs.

CI's command file features provide many useful capabilities to improve ease of operation and to increase productivity. The features covered here included user-defined variables, pre-defined system variables, return string and return status variables, and command file control structures.

1025. USING AN HP1000 FOR ROBOT COMMUNICATION

Vernon R. Sturdivant
Southwest Research Institute
6220 Culebra Road
San Antonio, Texas 78284

1. INTRODUCTION

The Cincinnati Milacron industrial robots in use today have been designed primarily for stand alone operation. That is, they are used without a separate computer. The robot is manually taught its operational sequence with the coordinate points being stored within the robot's own memory. These sequence points can even be complex enough to include decision points that cause conditional execution of a subsequence based upon an external condition. However, the conventional use of these robots has been in this stand alone mode.

The Robotic Deriveter System being designed and constructed for the Navy required the control and coordination of a large number of subsystems including a vision system and a complex end effector. In order to accomplish the control tasks, an HP A700 host computer was selected. It was desired for the robot to function as a computer peripheral whose task is to position the end effector based upon end effector and vision system measurements. An optional REMOTE function was purchased from Cincinnati Milacron that allows host computer communication with the robot via a three layer protocol. These protocol layers provide for error checking and retransmission of messages to insure error free communication.

This paper will describe the robot communication protocol layers and how they are implemented on an HP A700 computer for the Navy deriveter project.

2. ROBOT COMMUNICATION PROTOCOL

The Cincinnati Milacron ACRAMATIC Version 4 robot controller communicates with a host computer through a three layer protocol: a physical layer, a link layer, and an applications layer. Each layer provides services to the layer above. Services from one layer are provided by routines in that layer as well as the layer below. The physical layer provides a physical connection for the transmission of data by use of RS-232 serial communications. The link layer provides for the transmission, error checking, and retransmission of individual packets of data through the use of the Digital Equipment Corporation (DEC) Digital Data Communications Message Protocol (DDCMP). The applications layer provides the services to send commands to the robot and receive data from it. The applications layer protocol is a special one defined by Cincinnati Milacron.

2.1 Robot Operation

Normal robot operation involves execution of commands from its own memory. These commands are executed sequentially much the same as program execution in a digital computer. A robot program is referred to as a sequence with the commands contained in a sequence referred to as points. A point will contain coordinate data for robot movement and other information about attributes of the point. One attribute that is important for communication with a host computer is the REMOTE function.

2.2 Application Layer

Whenever the robot executes a REMOTE function within a sequence, the robot sends a Begin Remote Activity message to the host computer and waits for an Activity Request message from the host computer. At this point the host computer is in control and can send multiple Activity Requests to the robot during this one function execution. The robot performs each activity before receiving the next Activity Request. The Activity Requests implemented for the deriveter are:

(1) Send Sequence Data to the Host Computer
(2) Receive Sequence Data from the Host Computer
(3) Send Coordinate to the Host Computer
(4) Receive Coordinate from the Host Computer

The two activities concerned with sequence data allow the transfer of complete robot sequences between robot memory and host computer memory. The two activities concerning coordinate data allow reading the current robot coordinates or moving the robot to a given set of coordinates. The receive coordinate activity is the only activity that causes immediate robot motion.

To exit the REMOTE function, the host computer sends an Activity Request of No Activity to the robot, and the robot responds with an acknowledgement of the No Activity message. The robot is now released from the host computer and continues on to the next sequence point in its current sequence.

There are four basic types of communications messages used in the REMOTE function protocol. These message types together with the information fields within each type are as follows:

(1) Begin Remote Activity Message (Robot to Host Computer only)
    Message Identification - Two bytes.
    Robot Mode and Communications Status - One byte.
    Activity Type - One byte.
    Current Sequence Number - One byte.
    Current Point Number - Two bytes.
    Variable Data - 0 to 249 bytes.
(2) Remote Activity Request Message (Host Computer to Robot only)
    Message Identification - Two bytes.
    Activity Type - One byte.
    Variable Data - 0 to 253 bytes.
(3) Data Packet Message
    Message Identification - Two bytes.
    Activity Status - One byte.
    Variable Data - 0 to 253 bytes.
(4) Packet Request Message
    Message Identification - Two bytes.
    Activity Status - One byte.
    Variable Data - 0 to 253 bytes.

2.2.1 Send Sequence Data to the Host Computer

The Send Sequence Data activity instructs the robot to send one or more points of a selected sequence to the host computer. To initiate the Send Sequence Data activity the host computer sends a Send Sequence Data Activity Request. The robot responds by sending the first Data Packet containing the first sequence data point. The host computer continues by sending a Packet Request message. Sequence data points are sent in this fashion until the requested number of points have been sent.

2.2.2 Receive Sequence Data from the Host Computer

The Receive Sequence Data activity instructs the robot that it is to receive a new sequence from the host computer. The points received replace existing points. The host computer sends a Receive Sequence Data Activity Request to initiate this activity. The robot responds by sending the first Packet Request message. The host computer then sends the first Data Packet containing the first sequence data point. The host computer continues sending data points until the requested number of points have been sent.

2.2.3 Send Coordinate to the Host Computer

The Send Coordinate activity returns the current real world coordinate position and orientation of the robot. After the host computer sends the Send Coordinate Activity Request message, the robot responds with a Data Packet containing the current coordinate position of the tool center point and the orientation angles of the tool centerline.

2.2.4 Receive Coordinate from the Host Computer

The Receive Coordinate activity causes the robot to position and orient the tool to a new set of real world coordinates and angles. As stated earlier, this is the only REMOTE function activity which causes the robot to move. The host computer first sends a Receive Coordinate Activity Request message containing the new robot coordinates. After the robot motion is completed, the robot sends a Data Packet indicating that the activity has been completed.

2.3 Link Layer

The link layer protocol is DDCMP. Only DDCMP control and data messages using full duplex communication between two nodes have been implemented for the deriveter project.

2.3.1 Data Messages

Numbered data messages carry the application protocol messages over the DDCMP link. The message number assures correct message sequencing by the receiver. Data messages contain a header of fixed length and data of variable length. The data length information is contained in the COUNT field within the header. All data messages start with the ASCII SOH character. Data messages contain the following fields:

(1) Message ID - ASCII SOH message identification character (one byte).

(2) COUNT - Length of the data information (two bytes).
(3) RESP - Response number to acknowledge correctly received messages (one byte).
(4) NUM - Message number of this message (one byte).
(5) ADDR - Not used. Set equal to one (one byte).
(6) Header CRC - Header block check (two bytes).
(7) Data - Data portion of message (COUNT bytes).
(8) Data CRC - Data block check (two bytes).

2.3.2 Control Messages

Unnumbered control messages carry channel control information, transmission status, and initialization notification between nodes. All control messages begin with the ASCII ENQ character. Control messages contain the following fields:

(1) Message ID - ASCII ENQ message identification character (one byte).
(2) TYPE - Control message type (one byte).
(3) SUBTYPE - Control message subtype (one byte). This field has meaning only for NAK messages.
(4) RESP - Response number to acknowledge correctly received messages (one byte). This field has meaning only for ACK and NAK messages.
(5) NUM - Number of last sequential numbered data message sent (one byte). This field has meaning only for REP messages.
(6) ADDR - Not used. Set equal to one (one byte).
(7) CRC - Message block check (two bytes).

2.3.2.1 Start Message (STRT)

The STRT message (TYPE=6) is used to establish initial control and synchronization on the DDCMP link. It is used only on link startup or reinitialization. It operates with the start acknowledge message described below. The start sequence resets the data message numbering at the robot and host computer.

2.3.2.2 Start Acknowledge Message (STACK)

The STACK message (TYPE=7) is returned by the robot in response to a STRT from the host computer when the robot has completed initialization and reset data message numbering.

2.3.2.3 Acknowledge Message (ACK)

The ACK message (TYPE=1) is used to acknowledge the correct receipt of numbered data messages. It conveys the same information as the RESP field of numbered data messages.

2.3.2.4 Negative Acknowledge Message (NAK)

The NAK message (TYPE=2) is used to pass error information from the data receiver to the data sender. The NAK message also includes the same information as the ACK message, thus serving two functions: acknowledging previously received messages and notifying the sender of some error condition.

2.3.2.5 Reply to Message Number (REP)

The REP message (TYPE=3) is used by the robot to request received message status from the host computer. It is sent when the robot has transmitted data messages and has not received a reply within a time-out period. The response by the host computer is either an ACK or NAK depending on whether the host computer has or has not received all the messages previously sent by the robot as determined by the last robot message number sent in the NUM field of the REP message.

3. IMPLEMENTATION

3.1 Application Layer

The programmatic interface to the application layer is through one routine with seven operational modes. This routine is written in Fortran 77 and has the following calling sequence:

```
CALL ROBOT(MODE,BUF,LEN,IERR)
```

where MODE is a character constant specifying one of the seven modes:

(1) LINK - Initiate the linkup process.
(2) START - Receive the robot Begin Remote Activity message.
(3) MOVE - Robot receives coordinate message and moves to the specified coordinates.
(4) READ - Robot sends current coordinates.
(5) READ_SEQ - Robot sends sequence data.
(6) SEND_SEQ - Robot accepts sequence data.
(7) STOP - Sends No Activity message to robot to terminate REMOTE function. Robot proceeds to next point in its current sequence.

BUF is data to be sent to robot or received from robot, LEN is the length of BUF, and IERR is an error parameter returned to signal whether the mode successfully completed or not.

Basically, each mode consists of sending the robot the Activity Request message corresponding to the required mode and then reading the robot response. The messages to the robot are encoded and the messages from the robot are decoded by routines written in assembly language to allow easy byte manipulation. In order to provide error recovery, each message received from the robot is checked for the following features in the order given:

(1) Is it a data message? - Does it contain application layer information? If it does, skip to item (4).
(2) Is it an ACK message? - It should be a data message. If it is an ACK message, ignore the ACK and repeat the read to obtain the required information. Then check the new received message.
(3) Is it a NAK message? - It should be a data message. If it is a NAK message, repeat the Activity Request message to the robot and repeat the read to obtain the required information. Then check the new received message.

(4) Does it have the expected message ID? - Application layer message identification code (ID) must match that of required message. If it is a Begin Remote Activity message ID, the robot and host computer are not in step so repeat the Activity Request together with the read for the required information.

The details of the implementation of each mode are discussed below.

3.1.1 LINK Mode

The LINK mode allows synchronizing the robot and host computer message numbering. Also, this mode is used to initialize the serial multiplexer port by flushing the type-ahead buffer (Control Request with 26B function code).

3.1.2 START Mode

In the START mode the host computer waits for the robot seven byte Begin Remote Activity message that it sends when it reaches a REMOTE function point in its current sequence.

3.1.3 MOVE Mode

The MOVE mode consists of scaling the move coordinates to robot units and encoding the coordinates into the seventeen byte Receive Coordinate Activity Request. The Send Data Message link layer routine is used to transmit the request to the robot with the link layer Read Message routine used to receive the robot reply. The receipt of a No Activity Response indicates that the robot has completed the move.

3.1.4 READ Mode

The READ mode consists of sending the three byte Send Coordinate Activity Request message and then reading the robot coordinates using the link layer Read Message routine. The received message is decoded into the robot coordinates using an assembly language routine and then is scaled to engineering units.

3.1.5 READ SEQUENCE Mode

The READ SEQUENCE mode consists of sending the nine byte Send Sequence Data Activity Request message and then looping to read each successive sequence point in the robot Sequence Data Packet message until the robot signals the end of sequence by sending an activity status of zero in the last Data Packet message.

3.1.6 SEND SEQUENCE Mode

The SEND SEQUENCE mode consists of sending the eleven byte Receive Sequence Data Activity Request message and waiting for the three byte Sequence Data Packet Request message from the robot. When this message is received, the sequence data for the first point is scaled to robot units and coded into a twenty-two byte Sequence Data Packet and transmitted with the link layer Send Data Message routine. Each time the robot sends the Sequence Data Packet Request message, the next sequence point is sent until all points have been sent. The robot is signaled that the last point has been sent by setting the activity status bit

to zero in the last data packet. The robot responds with a three byte Sequence Data Packet Request message with the activity status bit also set equal to zero.

3.1.7 STOP Mode

The STOP mode is used to signal the robot the end of the REMOTE function and allow it to continue its current sequence. A three byte No Activity Request message is sent to the robot for this purpose.

3.2 Link Layer

The link layer top level software is implemented by three routines: two to send messages and one to receive messages.

3.2.1 Sending Messages

Two separate routines are used to send DDCMP messages to the robot: one for sending control messages and one for sending data messages. The sending of a control message consists of encoding the ID, type, subtype, response number, message number, and address fields; computation of the CRC for the encoded fields; the merging of the framing bytes, the encoded fields and the CRC; and an EXEC call to transmit the message to the robot. The encoding of message fields and the merging of the complete message are performed by assembly language routines.

The sending of a data message consists of encoding the ID, count, response number, message number, and address fields into the message header; the computation of the CRC for the header; the computation of the CRC for the data; the merging of the framing bytes, the encoded fields, the CRC for the header, the data, and the data CRC; and an EXEC call to transmit the message to the robot.

Framing bytes containing hexadecimal FF are added at the beginning and end of control and data messages to aid in synchronization (See Reference 2).

3.2.2 Receiving Messages

Only one routine is used to receive messages from the robot since it is not possible to know ahead of time if a message is a control message or a data message. Because the length of the control message and the header portion of a data message are the same length, the initial task in reading any message is to read only the first eight bytes (ignoring any framing bytes). The control and data messages can be separated based upon the first byte: ENQ for control messages and SOH for data messages.

If the message is a control message, it is checked for re-linking (STRT) or reply to message number (REP). If it is a STRT message, re-linking is performed and another attempt is made to read the required information. If it is a REP message, the NUM field is checked against the last good received message number (LGRMN) and if they match an ACK is sent. Otherwise a NAK is sent.

If the message is a data message, the length of the data in the the message is obtained from the COUNT field of the header and another read is made to obtain the data. An ACK is always sent to the robot in response to the receipt of the

data.

Messages sent by the robot are held in the type-ahead buffer in the A700 serial multiplexer so that messages can be received at any time. Due to the necessity of making two reads to obtain a complete data message, it was necessary to write a routine to manage the received information. This routine maintains an internal buffer of received information that has not yet been requested by the message read routine. Any time a request is made for a message, this internal buffer is checked for the required number of bytes. If they are contained in the internal buffer, they are sent to the message read routine and removed from the internal buffer. If an insufficient number of bytes are contained within the internal buffer, the type-ahead buffer is terminated (Control Request with 52B Function Code) and the available information there is read into the internal buffer. If the type-ahead buffer is empty at the time when the buffer is terminated, the routine waits at the read instruction for information from the robot and then obtains it one byte at a time.

The HP12040B serial multiplexer for the A-series computers allows partial reading of the type-ahead buffer without clearing the remaining information. However, these routines were initially written for and checked out using an HP12792B serial multiplexer and an E-series computer which does not provide for partial reading of the type-ahead buffer.

3.2.3 CRC Generation and Checking

The DDCMP protocol employs the CRC-16 polynomial ($X^{16}+X^{15}+X^{2}+1$) to generate the block check bytes for control messages, data message headers, and data message data fields. The technique employed to compute the block check bytes emulates a hardware shift register with modulo-2 add feedback to bits 2, 15, and 16. This routine is written in assembly language and takes advantage of the A700 shift and exclusive-or (modulo-2 addition) instructions by storing the current CRC value in the A-register and the new data byte in the B-register.

A received message can be checked by computing the CRC over the message together with its block check. If there has been no error made, the resulting CRC will be zero. The same routine is used for both generating and checking the block check fields.

3.2.4 State Variables

State variables for the communication channel are kept in COMMON for access by all link level routines. These variables are:

(1) LSMN - This is the last sent message number. It is used to number data messages and is incremented after each data message is sent.
(2) LGRMN - This is the last good received received message number. It is updated each time a data message is received and is checked whenever a REP control message is received to determine if any messages have been lost.

4. CONCLUSIONS

The routines described in this paper have proven to provide reliable communication

with the robot. The control functions provided have accomplished all of the deriveter project requirements even though they are a subset of those available from Cincinnati Milacron. In fact, only three basic operations are actually used: (1) move the robot to given coordinates, (2) read the robot's current coordinates, and (3) read a sequence from the robot.

5. ACKNOWLEDGEMENT

This work was supported under U.S. Navy Contract No. N00244-38-C1281.

6. REFERENCES

(1) RTE-A Driver Reference Manual, June 1983, Manual Part No. 92077-90011, Hewlett-Packard Company, Cupertino, California 95014.

(2) DECNET DIGITAL NETWORK ARCHITECTURE, Digital Communications Message Protocol, DDCMP Specification Version 4.0, March 1978, Order No. AA-D599A-TC, Digital Equipment Corporation, Maynard, Massachusetts 01754.

(3) Supplemental Communications Manual for Cincinnati Milacron ACRAMATIC Version 4.0 Robot Control, November 1983, Publication No. 10-IR-8332-S, Cincinnati Milacron, Cincinnati, Ohio 45209.

1026. PROGRAM DEVELOPMENT IN A LARGE HP/1000 NETWORK

Paul F. Gerwitz
Management Services Division, KP
Eastman Kodak Company
1669 Lake Ave
Rochester, NY14650

INTRODUCTION

In designing and implementing a large manufacturing network, particular emphasis must be put on the application code and it's long term effects on supportability. Items such as standard coding, use of common subroutines, predefined data types and record formats and methods of source module revision tracking were considered and implemented as part of the network development specification. This paper will describe these major elements that were used in such a network. The paper describes the network configuration, goals for implementing the application code and systems, methods of revision tracking and overall standardization techniques.

NETWORK DESCRIPTION

Application of computer technology has been for many years a prime activity at Kodak Park. The Application of computer systems spanning micro's, minicomputers and mainframe systems are an integral part of many manufacturing operations. One of the newest products to utilize computer technology in the manufacturing environment is the network for Kodak Disc Film. The original design of this project dictated a highly visible and integrated information system which would provide several levels of data collection and process monitoring through the use of computers spanning the spectrum of size and functionality. A network was proposed and finally implemented to fill these needs and provide flexibility for future expansion. This network consists of a 4 level hierarchical architecture with Programmable Logic Controllers (PLC) at the lowest level and an IBM mainframe system at the top. The PLC's are primarily responsible for production machine control and data collection. There are also small micro computer systems which provide specific functions (bar code readers, label printers, etc). The HP/1000 systems occupy the 2nd and 3rd levels of the network and provide production data storage and process monitoring control and reporting. This part of the network consists of 35 HP/1000 E-series processors divided into 3 functional groups, Plastic and Metal Parts manufacturing, Film Finishing and Program Development and Testing. Communications to the PLC's and micro computers are provided through vendor software products and in-house hardware/software where vendor solutions were not available. The HP/1000 systems communicate via DS/1000-IV links. The IBM mainframe system receives the data from the HP/1000 network via an RJE link and provides report and analysis functions as part of a larger Kodak Park database system. The remainder of this paper will describe the HP/1000 part of the network.

HARDWARE AND SOFTWARE CONFIGURATION

Configuration of each of the HP/1000 systems was dependent on a predefined set of criteria. These are:

1. Maintenance of all HP/1000 hardware must be performed by the in-house maintenance organization.

2. Production machine uptime must be maximized.

3. All system debugging and hardware repair must be performed off-line.

4. Parts and backup hardware must be available for immediate use when the need arises.

Based on the criteria, the hardware was configured as follows:

1. The processors are E-Series processors with equal amounts of memory. Initial memory consisted of 256kw, additional memory was added across the network as needed. Also Fast Fortran Microcode was added to the processors later to improve performance primarily for .ENTR due to heavy use of software subroutines.

2. All systems require I/O extenders, to accomodate the large number of devices and DS/1000 links.

3. All systems use a 7906 MAC controller disk as the system disk unit. 7925 disk drives are used on those systems requiring additional disk memory. All systems use a standard disk subchannel layout to facilitate interchangability.

4. The backplane layouts in both the main processor boxes and the I/O extenders are standardized as much as possible. Such interfaces as the Time Base Generator, disk controller, System Console and DS/1000 links were arranged using a standard select code map. Unused select code locations on a given processor are filled with jumper cards. Groups of select codes are reserved for designated device types or interface card types.

5. All systems are connected via DS/1000 for primary communication and have at least 1 redundant link in case of primary link failure.

6. Backup hardware (configured processors, I/O extenders, peripherals) are maintained for immediate use. Backup system disk packs are stored in the same area as the production computer.

All HP/1000 systems in the network are currently running revision 2301 of RTE-6/VM. The network was first installed running RTE-4B and converted during 1983. All application code is written in Pascal or Assembler. Data storage is handled through IMAGE/1000 and an in-house Database Manager that provides capabilities which IMAGE did not provide at the time such as auto mirror image backup, multiple programs accessing one database at a time, remote database access transparent to the user. As stated before, network communication is handled through DS/1000-IV with the RJE/1000 utilized for communication to the IBM mainframe. In addition to the use of HP software, products from Corporate Computer Systems are also used. Source Control System (SCONS) is used for source archival, Text Formatter (TFORM) is used for documentation, SCREEN/1000 is used for terminal dialogue on some systems and SORT/1000 which provides basic sort

capability.

DEVELOPMENT AND TEST SYSTEMS

A focal point of the HP/1000 network is the development system. This system is configured to provide on-line development capability for approximately 30 programmers. The system consists of an E-series processor with 2mb of memory, 404mb of disk storage (3 7925's, 2 7906's), 20 terminals, a 2608 line printer, mag tape unit and a 9872 4 pen plotter. The primary function of this system is to provide editing, compiling and source archival for the rest of the network. No special devices are included and program testing is kept to a minimum.

There are also 2 test systems present in the network. Both follow the same disk and backplane layouts as the production computers. Test 1 includes interfaces and devices which can be found on the production processors, providing the capability to test code with actual hardware in an off-line environment. Test 2 has only a processor, disk and terminal. Both systems are also used to perform program loading for subsequent installation on a production system. The system installation procedure will be described later.

GOALS FOR PROGRAM DEVELOPMENT AND IMPLEMENTATION

When the project was in the initial design stages, a firm set of guidelines were established and was used to guide the implementation strategy. The goals were modified as the project matured and the team gained experience with the technology.

1. Code should be written in one high level language to ensure long term supportability. Assembler will be utilized in cases where the high level language does not provide adequate capabilities. These included access to system tables, drivers etc.

2. Code should utilize structured design and coding techniques.

3. Code should utilize standard coding techniques including standard abbreviations, external subroutine calling sequences, types etc.

4. A procedure must be established to track application code changes to allow support of multiple program versions.

5. Support of a large development team must be maintained. Services such as concurrent editing and compiling environments, management of large amounts of disk space, communication of common procedures and coding techniques and support of the hardware and software technology as far as using it effectivly in design and implementation.

6. Schedules must be met for system startup and installation and software development and support costs must be minimized.

IMPLEMENTATION OF GUIDELINES

This section describes the implementation steps used to fulfill the guidelines that are stated above. These steps are organized by groups that specify general areas of concern. Appendices are referenced for additional detail.

PROGRAM STANDARDIZATION

The project team evaluated the two high level languages available at the time, FORTRAN 4 and PASCAL. PASCAL was chosen because of its structured syntax, code readability and user defined typing. Key members of the project team were part of a beta test of the original HP version of Pascal and found it to be well suited to our needs. Some initial programs were written in FORTRAN while the beta testing was going on and were compared to corresponding PASCAL versions, the Pascal versions demonstrated clearly the benefits for code readability and structure. The only significant problems with using PASCAL are the large memory requirement and slow compilation time compared with FORTRAN. Memory capacity on the Development system was increased to 2 MB, the maximum allowed, and the compiling was and is still run through the batch system. On-line compiling is permitted only in situations when a serious problem exists in a program which is affecting production. When the development system was upgraded to RTE-6/VM, the new version of the compiler doubled the compile speed, which has greatly improved programmer effectiveness.

The project did not have the luxury of Symbolic Debug and had to revert to coded debug statements, tracing capability etc. Some users with familiarity with the assembler debug subroutine DBUGR were able to use it successfully with PASCAL. Unfortunately the majority of programmers reverted to the tedious trial and error method of debug. Other procedures such as code walk throughs, structured design specifications and paper debug techniques were employed to reduce the use of the trial and error method.

PASCAL include files are used extensively throughout the code. Users bring these include files into the source program by inserting a $INCLUDE compile directive in the source code. These include files are of two distinct types. System Global Includes contain standard constants, data types, and record structures. Files were built for subsystems such as DS/1000, IMAGE/1000, FMP data types as well as system related data types such as EXEC call option bits and standard buffer and character string types. These provided an excellent way for users to standardize large sections of code.

The second type of includes are termed Subroutine Includes. Since all external procedures in PASCAL must be declared before they are used, these includes provide an excellent way of standardizing the calling sequence to externals. Each include contains a subroutine name of up to 80 characters in length although in practice the length was up to 20 characters. The parameters are then listed followed by an $ALIAS directive and EXTERNAL directive. Some of the subroutine calls would have multiple formats or names if there was a clear need to distinguish differences. See Appendix C for examples of include file usage.

The project team also made extensive use of pre-defined skeleton files for source and documentation files. Source header files contain the program name, author,

a description and a descriptive block called a revision log. For Pascal programs, compiler options are included such as $RANGE OFF$, $HEAP 0$, $RECURSIVE OFF$, $TITLE and $SUBTITLE. For assembler code a HED line is also included. These compiler options were selected as the defaults to prevent additional code for range checking, heap/stack management etc to be pulled in from the Pascal Library. If users need some of these options, they could be specified via the Pascal Options file in compiler runstring. These skeleton files provide a building block for the programmer when writing new programs.

The most essential component of the header is the $PASCAL or NAM record. This record contains the release.revision of the module along with the comment that would eventually appear in the relocatable file once the module is compiled. This provided the central means of tracking source code changes. The records used for Pascal and Assembler are:

```
$PASCAL 'COMMENT GOES HERE      %rel13%.%rev03%'  {<850731.1455>

      NAM PNAME,7 ---put comment here--- %rel13%.%rev03%<850731.1455>

'''/''''1''''/''''2''''/''''3''''/''''4''''/''''5''''/''''6''''/
```

The %rel13%... field are keywords used by the Source Control System for the current release/revision. The comment field data is placed in the relocatable file after compilation and the compile timestamp from Pascal is appended to the end. The edit timestamp is not transferred to the relocatable because it is enclosed in comment braces. For assembler files, the edit timestamp is placed so that it will line up with the other modules in the load. This record contains all the information necessary for a user to trace back a program to it's original source file. How this record is utilized will be explained later in this paper.

STANDARD SUBROUTINE LIBRARY

Subroutine libraries are a very common method used to reduce coding and provide a central pool of software for all programmers. The project makes extensive use of subroutine libraries. Specific application functions use libraries, even extending across application systems. The largest library is referred to as $SLIB or the System LIBrary. This library consists of code to access system table information, conversion utilities such as real to ASCII, integer to ASCII, standard interface utilities to IMAGE and DS/1000 and terminal I/O interfaces. Responsibilty for this library has now been transferred to a central support group where it is now available to any user within the company.

The subroutines in the library follow the same coding standards that apply to application code. But the subroutines need to be written to execute as fast as possible and to use as little code space as possible. Since Pascal generates fairly efficient code, most subroutines are written in Pascal. Assembler is utilized for specific functions where Pascal is clearly lacking, such as number conversion, id segment modification, system linked list traversal and i/o device table modification.

ITERATIVE CONSTRUCTS

To make optimum use of the Pascal language syntax, standards were created to guide the programmer in the use of the various structured elements. These were called Iterative Constructs. They were created to provide a common method of coding, optimize the source code storage requirements and provide a common readable format for future modification. Members of the team spent a significant amount of time studying the code that was generated by the compiler to determine the optimum coding strategy. This study produced a set of standard coding syntax that would result in the most optimum code possible. Examples of this can be found in Appendix A.

To some, these constructs may appear a little hard to read. The common practice of placing BEGIN, END, THEN on separate lines was deemed unfeasible due to the enormous amount of disk space necessary to store lines that only contained these reserved words. Also by using these constructs the typical problem of 'climbing stairs' was minimized. This problem occurs when users indent blocks of code and use separate lines for BEGIN, END etc causing the code for a given logic block to extend too far to the right or across a listing page.

```
IF  xxxxxxxxxxxx
  THEN
     BEGIN
         .
         .
       IF yyyyyyyy
          THEN
             BEGIN

          code tends to move right --------->
          and down |  at a very fast rate
                   |
                   |
                   V
```

The method that was used helps to compress code without compromising readability.

```
IF  xxxxxxxxxxxx  THEN BEGIN
         .
         .
    IF yyyyyyyy THEN BEGIN
```

PROGRAM AND FILE NAMING CONVENTIONS

Naming programs in the RTE environment has proven to be a difficult task at best. But attempting to design a scheme for 4000 programs and segments is formidable indeed. The scheme used was that file names for program main modules, segments, and subprograms have the following format:

(prefix)(group)(id)(module id)

where

```
prefix     is one of the standard file prefix characters
group      is a 2 letter program group name (see below)
id         is the program id within the group (any 2 characters)
module id  indicates the type of module
              null for a main module
              0-9  for a segment (may use alphas if >10 segments)
              A-Z  for subprograms
```

```
AB12 could consist of main       AB12
                      subprogram AB12A
                      segment    AB120
                      subprogram AB12B
```

The 2 letter group name will indicate function type or data type. For example:
MD could mean Master Data,
IV could mean Inventory.

A similar problem exists for files in general. Since the project was forced to operate in the FMP file system, file naming was a very important task. It was quickly realized that without a consistent scheme, a development disk cartridge could quickly become very difficult to manage. Several solutions were proposed such as creating a master directory file on each cartridge or writing an in-house cartridge management package. These solutions were deemed to be impractical for this project. The naming scheme that was agreed to made use of the defacto list that HP has published in various manuals in addition to our own extensions. This list can be found in Appendix B.

SYSTEM BASED STANDARDS

SYSTEM UTILITY PROGRAMS

Utility programs were written to handle a variety of tasks including system error logging, class number management, remote spooling capability, source file cross reference utilities, type 6 file management etc. These programs are used throughout the network and are designed with the widest possible functionality. New programs continue to be added to this library and enhancements are made to the existing utilities to provide greater functionality.

TYPE 6 FILE TIMESTAMPING

The RTE Link program places a time stamp and other information into the end of the first record of the type 6 file. This feature along with the NAM record comment field described previously provides the means to track a given program revision. Using the load time stamp provided by Link, a user can refer to a load map and determine which relocatables went into a given type 6 file. The user can then determine the version of the source module in question by looking in the comment field of the module in the load map. The data contained in this field will directly relate to a version of the source by virtue of the SCONS keywords or the EDIT or Pascal time stamps. The tail end of the first record also contains a description field which the user has the option to set either

at load time or through a utility program. The common practice is to place the release/revision of the module, and the "release" of the application system that the program is loaded on such as:

```
SYS:01.00: 1.005  - These are in words 105-112
HP :01.00:  2340    in type 6 file record 1
 ^    ^      ^
 |    |      |
 |    |      |--- release.revision or HP rev of the relocatable
 |    |
 |    |-----------release of "system"
 |
 |---------------program type
```

PRODUCTION SYSTEM STANDARDS

Standardization was also applied to the production system itself. Procedures were implemented to assure minimum requirements on a system level. Each system uses standard WELCOM and initialization files (DS, for example), system utility programs and transfer files, etc. These files are used to link programs, provide standard interfaces to remote printing programs, RP system programs, initialize system utilities and set system time.

The disk layout standard resulted in providing many useful advantages. The layout of the 7906 drive is as follows:

```
+-------------------------------------------------+ >
|  subchannel 0     |    subchannel 1             |  >
|  264 tracks       |    528 tracks               |   > removable
|  lu 10            |    lu 11                    |   >
|                   |                             |  >
| application files, databases etc.....          | >
+=================================================+
|  subchannel 2 | subchannel 3 | subchannel 4    | >
|  256 tracks   |  256 tracks  | 284 tracks      |  >
|  lu 2 (sys)   |  lu 3 (aux)  | lu 12           |   >
|               |              |                 |    > fixed
| op system, FMP| track pool,  | spooling, log   |   >
| track pool    | Type 6 files | files, sys FMP  |  >
| some t6 files | only         | files           | >
+-------------------------------------------------+
```

This layout provided a great deal of flexibility. Placing the operating system on the fixed platter was initially received with some reluctance, but after implementing it on several production systems, it proved to be a very effective implementation tool. It also allows the same system generation to reside on different processors, with unique production data being stored on the removable. The removable platter also provided a method of storing test data and test program relocatables which could be used on any system in the network if needed. Any additional disk memory requirements were satisfied by installing a 7925 drive on a system. The subchannel layout for these disk drives are also the same, so that disk packs can be interchanged if needed.

An idea that evolved from the implementation of this layout was that of the 'Master Loading Disk' or MLD. This pack exists in one of three types. First is the 'Production MLD'. This contains relocatables of the programs that are actually running on the system. This pack is only used when the production release is built and is then saved for backup. The second type is the 'Pending MLD' which contains relocatables for the next release of the production software. This pack will eventually become the production pack. The third type is the 'Test MLD' containing test versions of programs. These separate packs are all managed by one member of a given application group by hand written logging procedures. In addition to the various MLD's, a group may have a test data disk pack as well. Common system software is maintained on the Software Group MLD and is maintained outside the project by the central support core group.

SOURCE AND CHANGE CONTROL SYSTEM

SCONS

The Source Control System (SCONS) is a software package developed by Corporate Computer Systems and is designed to aid in the management of software development projects. There was no package from HP that provided this capability at the time and many of the programmers on the team previously worked in an IBM programming environment where this type of change control was already in use. As the amount of software increased it became readily evident that some method was needed to control the tremendous amount of software for the project.

The source control system functions on the concept of "release" and "revision". A revision is a modification to a file which corrects or enhances it in a small way such as a bug fix. A release represents a major change in functionality or philosophy. A release file would be separate from any previous versions of the file and be supported in it's own right. Release and revision notation is specified by two integers separated by a period or decimal point. The first number specifies the release and the second, the revision of that release. SCONS stores the entire source file at a given release and additional files containing only the changes to the original file; these are known as the 'Base' and 'Delta' files respectively. When a given release.revision of a file is needed, SCONS takes the base file and applies all the revisions to that base from the delta files until the specified release.revision is reached.

Each application group is assigned an individual disc cartridge. It is on this cartridge that all files being used by that group reside including the SCONS library and associated files. Each programmer has his own individual system and SCONS logon. SCONS stores each file under it's control by copying it to an internal file with a unique name. An entry is maintained in the SCONS library that relates the logical program name (i.e. &MM01) to a physical file name (i.e.)G2XYZ). When a user wishes to modify an existing program, he must "check out" the file to himself. Only one user can have a given file checked out at a time, preventing multiple changes to be made concurrently by different people.

TYPICAL CHANGE CYCLE

When a user wants to make a change to an existing source file, the file is

checked out of SCONS by the user. The checkout command creates an FMP file with the logical name, i.e. &MM01. A subparameter specified at checkout tells SCONS not to substitute the release.revision number in the keyword field in NAM record. The file is now modified by the programmer and recompiled. This continues until the module is error free and ready to test. When the file is compiled the relocatable NAM record will contain the %rel3%.%rev03% keywords, thus designating this as a test relocatable. The program is then tested either on the test computer, a non-production mode application system or the development system. The testing and debug step may require additional changes and re-compiles.

After the program has been thoroughly tested and is ready for release, it is returned to the SCONS system. At this point the decision is made as to whether this change dictates a new release or a revision to an existing release. Once this has been determined, the file is returned to SCONS and either a new base file or a delta file is automatically created and an entry made in the SCONS directory. The next step produces an SCONS relocatable. SCONS has the capability of interfacing to user written programs to process any file under it's control. SCONS will build a copy of a specified release.revision and then schedule a user written program. A set of programs were written to take advantage of this capability so that programs could be compiled without taking the files out of the SCONS subsystem. During this step, SCONS is instructed to replace the keywords in the NAM record with the actual release and revision numbers and the program is compiled using the appropriate compiler. The resulting relocatable has the release.revision number in the NAM record and it can then be loaded on a production system. The tracking of this program is now assured. The program is then moved to the Pending MLD for eventual release and the compile listing is filed for reference. When the program is loaded, the release.revision number can be placed in the type 6 file description field and will also appear in the load map.

CONCLUSIONS

The procedures and standards described in this paper were deemed appropriate for this application network by virtue of it's perceived lifetime and the need to be able to support the ever changing needs of the manufacturing organization. We feel that the major goals have been achieved and that the network can be supported with the small programming staff that is available longterm. Many innovative solutions were designed to extend the technology to the fullest. The effort has produced not only an efficient, effective, reliable network but also has produced highly technical individuals who have gone on to other projects where the wealth of experience is paying off in other areas. We are also sharing some of the common software in new and existing projects as a way of improving the use of the HP/1000 technology.

APPENDIX A

ITERATIVE CONSTRUCTS

```
a) FOR range_of_values DO
      simple_statement;

b) FOR range_of_values DO BEGIN
       statement_number_1;
       statement_2;
       statement_number_three;
   END;

c) WHILE condition DO
      simple_statement;

d) WHILE condition DO BEGIN
      first_statement;
      second_statement;
      last_statement;
   END;

e) REPEAT
      statement_number_one;
      statement_number_2;
      statement_number_3;
   UNTIL condition_is_true;

f) IF condition THEN
      simple_statement;

g) IF condition THEN BEGIN
      statement_number_one;
      statement_number_two;
      statement_number_3;
   END;

h) IF condition THEN
      simple_statement
   ELSE simple_statement;

i) IF condition THEN BEGIN
      statement_number_one;
      statement_number_two;
   END ELSE BEGIN
      else_statement_number_one;
      statement_number_2;
      last_statement;
   END;

j) IF condition_1 THEN
      statement_1
   ELSE IF condition_2 THEN
      statement_2
   ELSE IF condition_3 THEN
      statement_3
   ELSE statement_4;

k) CASE variable OF
      value_1 : statement_1;
      value_2 : BEGIN
                   statement_2;
                   statement_3;
                   statement_4;
                END;
      value_3 : statement_number_5;
      OTHERWISE all_other_conditions;
   END;
```

APPENDIX B

FILE NAMING CONVENTIONS

Hewlett-Packard

&	Source, RTE Sysgen Answer files (&---GN), DB schemas
%	Relocatable
'	Compiler listings, Loader listings, DBDS Schema listings
	Use 'name' for loader listings if name < 5 characters
	Absolute load modules, RTE Sysgen output files (---GN)
	Help files for CMD program (e.g. CMD)
$	Searchable relocatable libraries
*	Transfer files (FMGR instructions with : prompts)
^	PASCAL assembly code; also DS Network definition files
"	Documentation files, Help files, DBDS Schema list files
	Accts answer files
/	RJE commands; also IBM JCL
\	Spare

Kodak

#	Loader command files, Merge command files
&&	System-wide include files
_	Keydump files to program terminal softkeys
<	QUERY XEQ command files, DBBLD input files
>	IMAGE type 2 Data Base files, type 1 root file
?	SBULD scratch files
[	Forms library (SCREEN/1000), BRUNO picture files
]	Forms data dictionary (SCREEN/1000), Test data
(	Spare
)	Source Control system file (SCONS)
.	Backup relocatables, library production relocatables
;	KCD source file prefix
-	RTE sysgen listing files (from the software group)
D	Data files associated with a specific program.
I	Include files for individual application programs
	Include files for FMP, IMAGE, and LIBRARY subroutine calls
	Include files for EXEC and DEXEC calls
J	Job files (for batch submission)
L	Error Log files
P	Remote Spooling Output files from PRT/OUTSP
Q	Query select file (QSELnn recommended)
S	User defined spool files
T	Text Formatter (TFORM) Input Files
X	Source file with expanded INCLUDE files (via XINCL)

APPENDIX C

GLOBAL SYSTEM INCLUDE FILE EXAMPLES

```
        $LIST ON$  { USE SCONS UPDATE PROCEDURES}

{&&SYS  Rel.Rev=  1.001,     Revisor= PFG                <850731.1455>
        ################################################################
        ########    GENERAL DECLARATIONS FOR ALL SYSTEMS     ########
        ################################################################
        Requires no prior INCLUDE files
        $LIST OFF$

CONST
   exec01 =  1;   exec02 =  2;   exec03 =  3;   exec04 =  4;
   exec05 =  5;   exec06 =  6;   exec07 =  7;
   exec09 =  9;   exec10 = 10;   exec11 = 11;   exec12 = 12;
   exec13 = 13;   exec14 = 14;   exec15 = 15;   exec16 = 16;
   exec17 = 17;   exec18 = 18;   exec19 = 19;   exec20 = 20;
   exec21 = 21;   exec22 = 22;   exec23 = 23;   exec24 = 24;
   exec25 = 25;   exec26 = 26;   exec99 = 99;

   no_abort = -32768;  {100000B}
   z        =   4096;  { 10000B}
   echo     =    256;  {   400B}

   x_bit    =   1024;  {  2000B}
   a_bit    =    512;  {  1000B}
   k_bit    =    256;  {   400B}
   v_bit    =    128;  {   200B}
   m_bit    =     64;  {   100B}

   no_wait    = -32768;  {100000B  No wait for class I/O, RNRQ, LURQ}
   save       =  16384;  { 40000B  Save bit for class I/O}
   no_dealloc =   8192;  { 20000B  No Deallocate bit for Class I/O}

   rte6vm = -17;         {System Codes for I.OPSY}
   rte4b  = -9;
   rteA   = -37;
   rteA1  = -45;

   {NOTE: This constant will be retained until 3/1/84}
          {de_alloc   =   8192;   ------  NOTE: Do NOT use this any more

TYPE
   int          = -32768..32767;
   positive_int = 0..32767;
   char_set     = SET OF char;

   string_2  = PACKED ARRAY [1..2] OF char;
   string_3  = PACKED ARRAY [1..3] OF char;
```

```
   string_8   = PACKED ARRAY [1..8] OF char;
   string_9   = PACKED ARRAY [1..9] OF char;
   string_10  = PACKED ARRAY [1..10] OF char;
   string_12  = PACKED ARRAY [1..12] OF char;
   string_66  = PACKED ARRAY [1..66] OF char;
   string_80  = PACKED ARRAY [1..80] OF char;

   prog_name_type = string_6;
   messs_type     = string_40;

   array_2   = ARRAY[1..2] OF int;
   array_3   = ARRAY[1..3] OF int;
   array_7   = ARRAY[1..7] OF int;
   array_8   = ARRAY[1..8] OF int;
   array_9   = ARRAY[1..9] OF int;
   array_10  = ARRAY[1..10] OF int;
   array_12  = ARRAY[1..12] OF int;
   array_16  = ARRAY[1..16] OF int;
   array_32  = ARRAY[1..32] OF int;
   array_128 = ARRAY[1..128] OF int;

   register_type = RECORD
                     CASE int OF
                        1 : (num : int);
                        2 : (str : string_2);
                   END;

   sys_time_rcd = RECORD
                    CASE int OF
                    1 : (tens_of_msec : int;
                         seconds      : int;
                         minute       : int;
                         hour         : int;
                         day_of_year  : int;
                         month        : int;
                         day          : int;
                         year         : int);

                    2 : (arr : array_5);
                  END;

   user_time_rcd = RECORD
                     CASE int OF
                        1 : (hour_min  : int;
                             sec_tmsec : int;
                             day_year  : int;
                             year      : int);

                        2 : (systime : integer);
                   END;

{-------------------------------------------------------------------}
{-----------  END OF GENERAL SYSTEM DECLARATIONS (&&SYS) ------------}
{-------------------------------------------------------------------}
$LIST ON$
```

```
        $LIST ON$  { USE SCONS UPDATE PROCEDURES}

{&&DSG  Rel.Rev= 1.000,     Revisor= DWR                <850731.1455>
        ###################################################################
        ########        DS/1000 GLOBAL DECLARATIONS          ########
        ###################################################################
        Requires &&SYS                                                   }
        $LIST OFF$

CONST
   ds_local_node = -1;     {For local DS node}

   ds_open_req    = 1;     {Function codes for GET call}
   ds_read_req    = 2;
   ds_write_req   = 3;
   ds_control_req = 4;

   ds_terminate_req = -1;

TYPE
  {++++++++++++++++++++++++++++++++++++
   + DS/1000 CONTROL/TAG FIELD TYPES   +
   ++++++++++++++++++++++++++++++++++++}

   ds_control_type = ARRAY [1..4] OF int;

   ds_tag_type     = ARRAY [1..20] OF int;

   ds_cartridge_type = RECORD
                          CASE int OF
                             1 : (num  : int;
                                  node : int);
                             2 : (str  : string_2);
                       END;

{-------------------------------------------------------------------}
{----------- END OF DS/1000 GLOBAL DECLARATIONS (&&DSG)  ------------}
{-------------------------------------------------------------------}
$LIST ON$


        $LIST ON$  { USE SCONS UPDATE PROCEDURES}

{&&FMP  Rel.Rev= 1.001,     Revisor= PFG                <850731.1455>
        ###################################################################
        ########  GENERAL FILE MANAGEMENT (FMP) DECLARATIONS ########
        ###################################################################
        Requires &&SYS                                                   }
        $LIST OFF$

CONST
   dcb_size_1  =  144;    {Standard DCB buffer sizes by # blocks}
   dcb_size_2  =  272;
```

```
    dcb_size_3  =  400;
    dcb_size_4  =  528;
    dcb_size_5  =  656;
    dcb_size_6  =  784;
    dcb_size_7  =  912;
    dcb_size_8  = 1040;
    dcb_size_9  = 1168;
    dcb_size_10 = 1296;

    exclusive_access = 0;  {File open access options}
    shared_access    = 1;
    update_access    = 2;

TYPE
  {++++++++++++++++++++++++++++++++++++++++++++
   + SECURITY CODE/CARTRIDGE REFERENCE TYPES +
   ++++++++++++++++++++++++++++++++++++++++++++}

   security_type = RECORD
                     CASE int OF
                        1 : (num : int);
                        2 : (str : string_2);
                   END;

   cartridge_type = RECORD
                      CASE int OF
                         1 : (num : int);
                         2 : (str : string_2);
                    END;

  {++++++++++++++++++++++++++++++++++++++
   + FILE NAME/SIZE TYPES                +
   ++++++++++++++++++++++++++++++++++++++}

   file_name_type = string_6;

   file_size_type = RECORD  {For CREAT and ECREA calls}
                      CASE int OF
                         1 : (number_blocks : int;
                              record_length : int);

                         2 : (ext_blocks : integer;
                              ext_length : integer);
                    END;

  {++++++++++++++++++++++++++++++++++++++++++++++++
   + CARTRIDGE STATUS ENTRY RECORD FORMAT (ISTAT) +
   ++++++++++++++++++++++++++++++++++++++++++++++++}

   fmt2_type = PACKED RECORD
                  lock_word      : 0..255;
                  cart_lu_num    : 0..255;
                  last_fmp_track : int;
```

```
                cart_ref_num    : cartridge_type;
                session_id      : int;
            END;

   cart_entry_type = RECORD
                        CASE int OF
                           1: (cart_lu_num     : int;
                               last_fmp_track  : int;
                               cart_ref_num    : cartridge_type;
                               lock_word       : int);

                           2: (fmt2 : fmt2_type);
                    END;

   cart_list_type = ARRAY [1..64] of cart_entry_type;

  {++++++++++++++++++++++++++++++++++++++++++++
   + RECORDS FOR NAMR PARSING FUNCTION (NAMR)    +
   ++++++++++++++++++++++++++++++++++++++++++++}

   namr_type = RECORD
                  CASE int OF
                     1 : (file_name : string_6;
                          parm_type : PACKED ARRAY [1..8] OF 0..3;
                          security  : security_type;
                          cartridge : cartridge_type;
                          file_type : int;
                          file_lng  : int;
                          rcd_lng   : int;
                          spare     : int);

                          2 : (device_lu : int);
                END;

{-------------------------------------------------------------------}
{----------- END OF FILE MANAGEMENT DECLARATIONS (&&FMP) ------------}

$LIST ON$


                SUBROUTINE INCLUDE FILE EXAMPLES
                --------------------------------

PROCEDURE read_from_disc (    request_code    : int;
                              function_and_lu : int;
                          VAR data_buffer     : ex01_buffer;
                              buffer_length   : int;
                              track_number    : int;
                              sector_number   : int)
   $ALIAS 'EXEC'$;
   EXTERNAL;
```

```
PROCEDURE read_from_device (    request_code     : int;
                                function_and_lu : int;
                            VAR data_buffer     : ex01_buffer;
                                buffer_length   : int)
   $ALIAS 'EXEC'$;
   EXTERNAL;

PROCEDURE read_from_driver (    request_code     : int;
                                function_and_lu : int;
                            VAR data_buffer     : ex01_buffer;
                                buffer_length   : int;
                                option_1        : int;
                                option_2        : int)
   $ALIAS 'EXEC'$;
   EXTERNAL;

{########################################################################}

PROCEDURE get_pgm_name (VAR pgm : prog_name_type)
    $ALIAS 'PNAME'$;
    EXTERNAL;

PROCEDURE open_data_base (VAR dbase_name  : data_base_type;
                              dbase_level : db_level_type;
                              access_mode : int;
                          VAR status      : data_base_status)
   $ALIAS 'DBOPN'$;
   EXTERNAL;

{########################################################################}

PROCEDURE open_ds_link (VAR ds_control_block : ds_control_type;
                        VAR error_code       : int;
                            slave_name       : prog_name_type;



                            slave_node       : int;
                            ds_tag_field     : ds_tag_type)
   $ALIAS 'POPEN'$;
   EXTERNAL;


PROCEDURE open_ds_clone (VAR ds_control_block : ds_control_type;
                         VAR error_code       : int;
                             slave_name       : prog_name_type;
                             slave_node       : int;
                             ds_tag_field     : ds_tag_type;
                             clone_option     : int)
   $ALIAS 'POPEN'$;
   EXTERNAL;

{########################################################################}
```

1027. IMPROVING CDS PROGRAM PERFORMANCE

Wayne R. Asp
Hewlett-Packard Company
2025 W. Larpenteur Ave.
St. Paul MN 55113

Under the RTE-A operating system, Code and Data Separation (CDS) gives the capability to execute programs larger than 64kb with excellent performance. Using CDS, many large FORTRAN programs are being written or converted from larger machines to run on the A-Series. The usage of CDS by these programs involves compiling the FORTRAN code with the CDS option, and then loading the program using automatic segmentation. MACRO routines can also be written to use CDS. While the first pass compilation and loading of these programs will run, it is usually possible to obtain much better performance. This performance can be gained by understanding the CDS implementation and taking advantage of CDS features, by analyzing and tuning the code segmentation, and by making trade-offs in performance and space requirements. This paper examines CDS features and performance issues for FORTRAN and MACRO programmers, and presents alternatives for implementation. As always, the user must evaluate the alternatives, select, and implement those that would be the most beneficial to any particular program.

The basic key to good CDS program performance involves understanding how CDS is implemented. Once the implementation scheme is understood, performance issues are more easily addressed. From a user's perspective, little must be done to convert a FORTRAN program from non-CDS to CDS. First, the program must be compiled under the directive '$CDS ON'. This instructs the compiler to emit relocatables with the code separated from the data. Next, the program is linked using automatic segmentation. When using automatic segmentation, LINK creates code segments by placing modules one-by-one into a segment. When the current code segment overflows 31 pages, the module creating the overflow is backed out, and the current code segment is considered complete. The backed out module is then placed into a new current code segment. This continues in a FIFO manner until all modules have been placed in code segments. The order in which the relocatable modules are presented to LINK affects the code segmentation which LINK performs. After linking, the program is ready to be run.

Converting a MACRO routine from non-CDS to CDS is a more involved task, and should only be attempted if absolutely necessary. When writing MACRO using CDS, it is the user's responsibility to account for the locations of all data in the MACRO routine; code, data, stack, or static. Because most non-CDS MACRO routines have their data and code interspersed throughout, the conversion to CDS is extremely difficult. Some guidelines for writing CDS MACRO routines are presented in Appendix C.

When the CDS program is run, two partitions must be allocated; one for the code, and one for the data (figure 1). The code partition is divided up into code blocks, each the size of the largest code segment created by LINK. There may be more code segments than code blocks, meaning not all code segments can be resident in memory at one time. When a code segment is executing, the segment is resident in a code block in the code partition. This code block is then mapped into logical memory in map set 3. The data partition is divided into

CDS Program Partitions

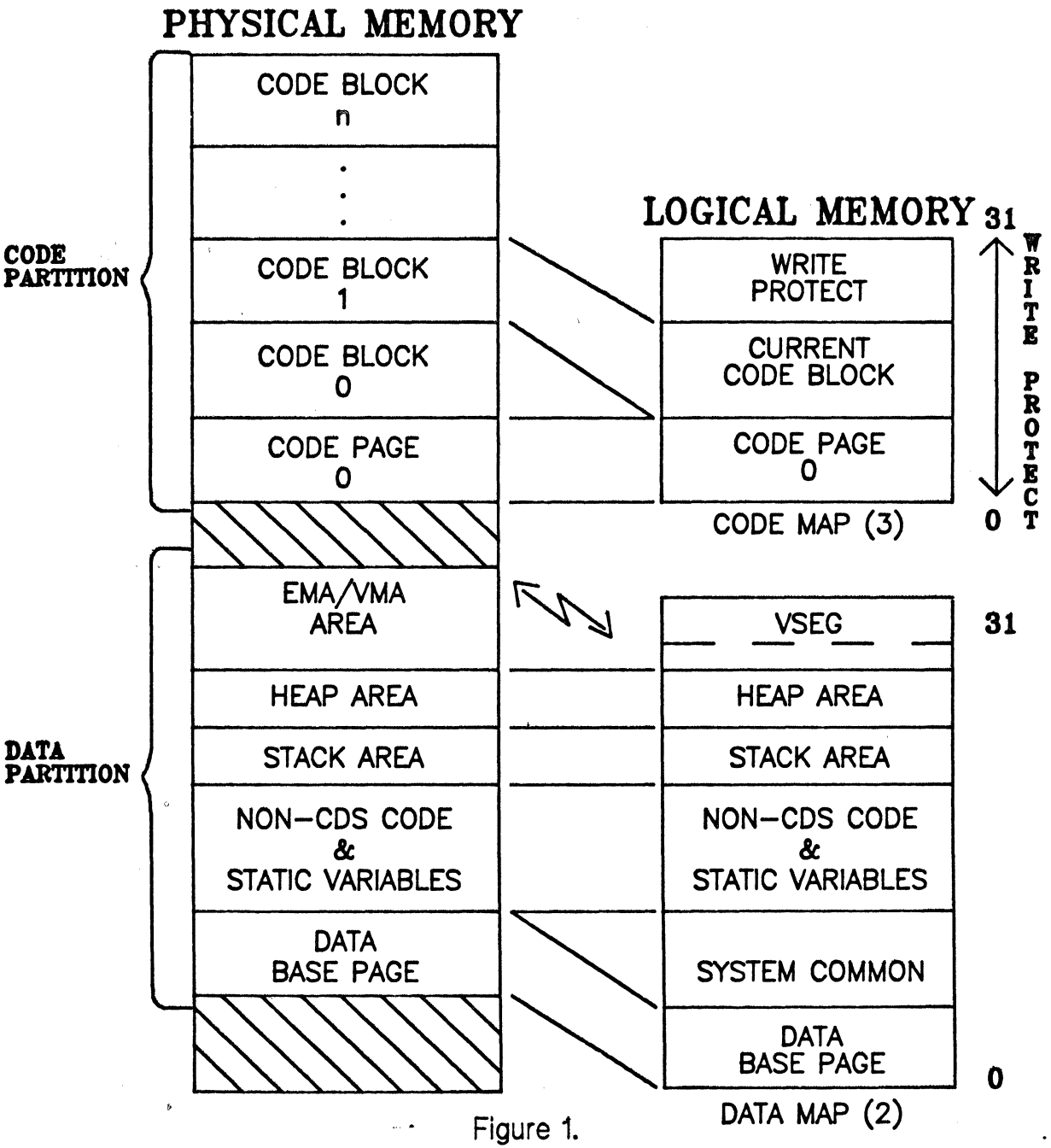


Figure 1.

several areas in physical memory. Any non-CDS code used by the program, plus the static variables for all of the CDS modules are resident immediately following the data base page. Next in the partition is the stack area. The stack is used to store dynamic variables and parameter addresses for CDS subroutines. Each time a CDS subroutine is called, a new entry is built on the stack. This entry is called a stack frame. When the subroutine is exited, the stack frame is popped. Following the stack area is the heap area. The heap can be accessed from FORTRAN and MACRO using the LIMEM system subroutine. Last in the data partition is the EMA/VMA area, if the program uses this feature. The data partition is mapped into logical memory using map set 2. System common is mapped into logical memory if the program was linked using system or labeled common.

Generally, programs linked using automatic segmentation will perform fairly well. In many cases, however, better performance can be gained by performing manual segmentation on the program. In addition, as the CDS programs get larger and larger, the CDS scheme becomes less and less transparent. The point at which this can occur varies from program to program depending on many factors: code size, data segment size, stack size, usage of EMA, etc. The presentation of two scenarios at this point will help to illustrate possible problems:

Scenario 1: A large FORTRAN program was converted to run on an A900. This involved moving a large data array into VMA. The converted program has 40 code segments of 20 pages each, and one data segment of 20 pages. Due to memory limitations, there are 15 code blocks in the code partition.

Problem: The program takes 24 hours to run on the A900. It should execute in less than 3 hours.

Scenario 2: A FORTRAN program is written to run on the A900. The program has 30 code segments of varying sizes, and one data segment of 31 pages. It uses System common and several 1K word arrays.

Problem: The program generates a CS06 error, meaning the stack has overflowed.

A brief discussion of the internal design of CDS is included here which will focus on limiting factors in the CDS design which can make CDS non-transparent to the user.

In CDS, the JSB instruction does not exist. This is because a JSB instruction saves the return address in the code of the called routine, which is illegal in CDS. The JSB type instruction in CDS is PCAL, or procedure call. The PCAL instruction loads the called code segment into memory (if necessary), maps the code block containing the code segment into logical memory (if necessary), and builds a stack frame for the called subroutine. PCAL also checks for stack overflow, resolves parameter addresses, and records the return address. There are different versions of the PCAL instruction. PCALI calls a subroutine in the current code segment. PCALX and PCALV call subroutines in a different code segment from the current one. This is called a cross segment call, since the called subroutine is in a different code segment from the caller subroutine.

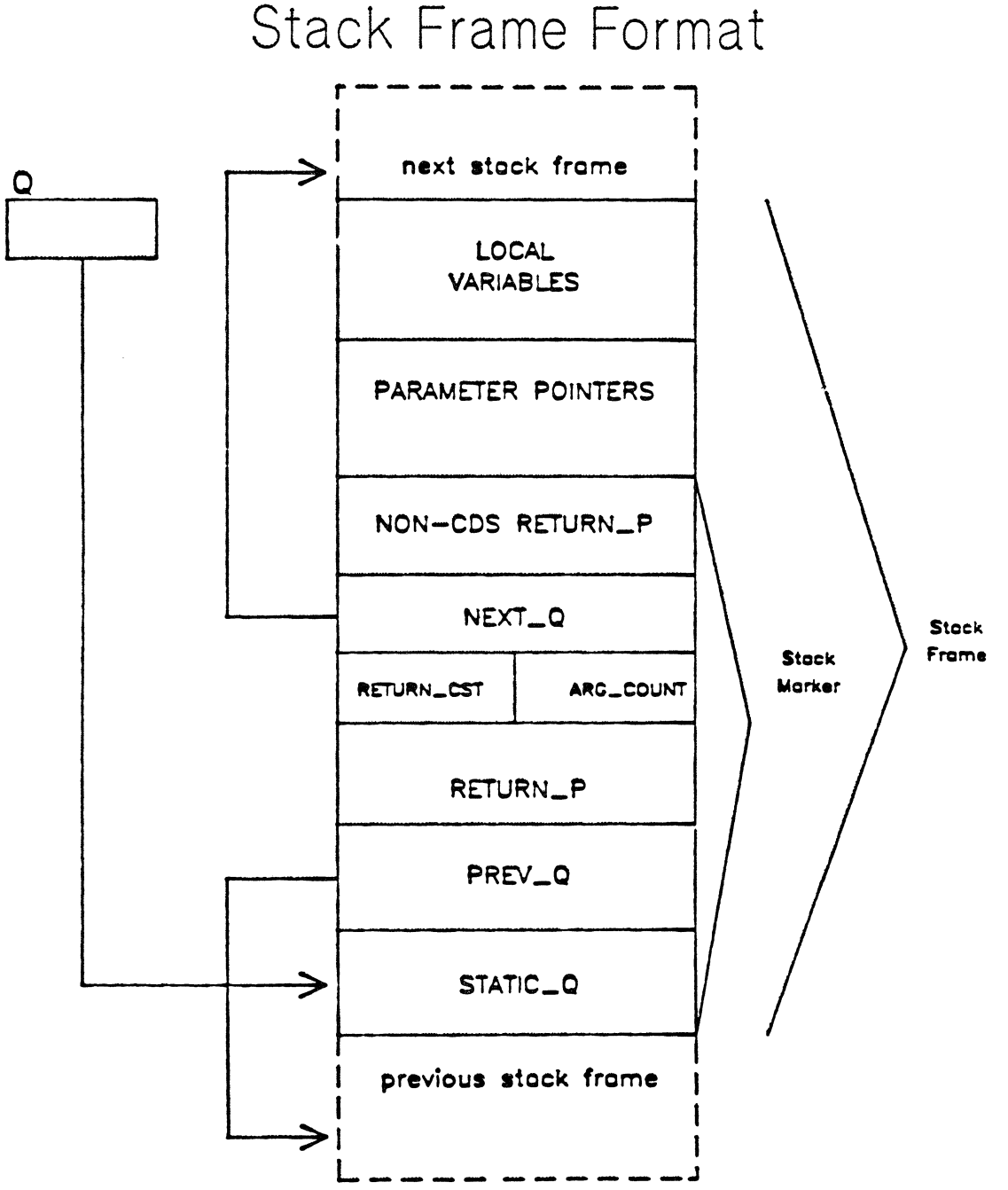
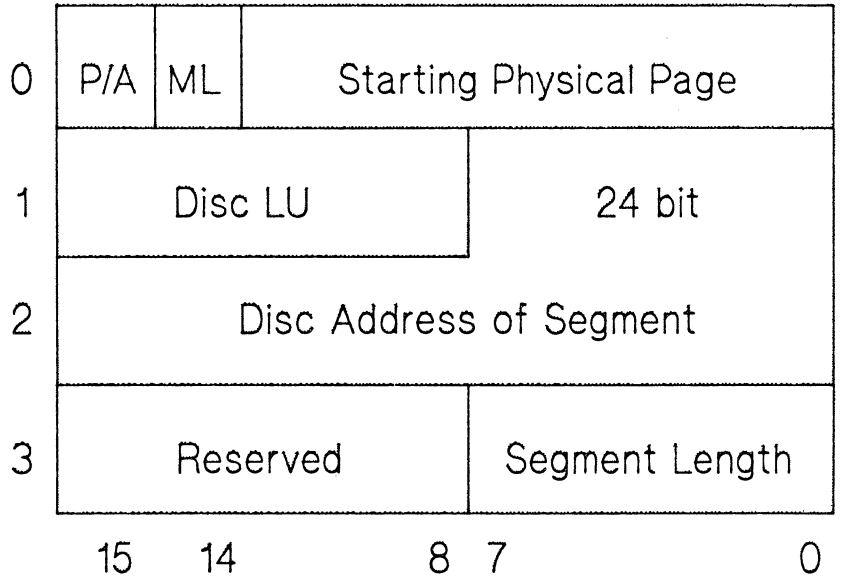

Stack Frame Format
Q
next stack frame
LOCAL VARIABLES
PARAMETER POINTERS
NON-CDS RETURN_P
NEXT_Q
RETURN_CST
ARG_COUNT
RETURN_P
PREV_Q
STATIC_Q
previous stack frame
Stack Marker
Stack Frame

Figure 2.

Code Segment Table Entry



P/A	=	0 IF THE SEGMENT IS IN MEMORY 1 IF THE SEGMENT IS NOT CURRENTLY IN MEMORY
ML	=	1 IF THE SEGMENT IS MEMORY LOCKED
DISC LU	=	LU OF THE TYPE - 6 FILE
DISC ADDRESS	=	OFFSET IN BLOCKS FROM THE BEGINNING OF THE DISC LU TO THE START OF THE CODE SEGMENT
CODE SEGMENT LENGTH	=	NUMBER OF 128 WORD BLOCKS IN THE CODE SEGMENT ON THE DISC

Figure 3.

Segment Replacement Table

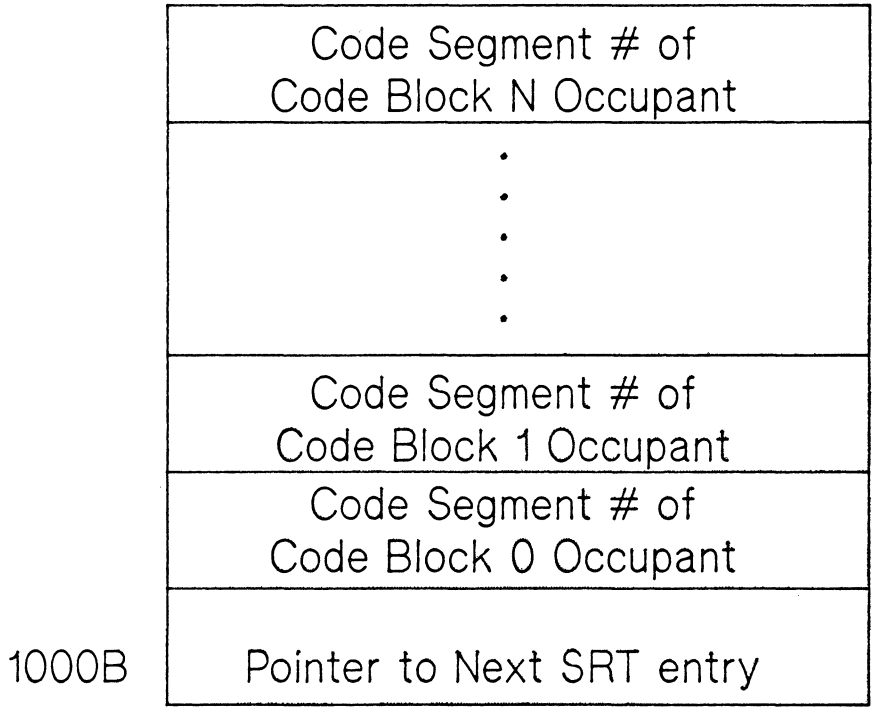


Figure 4.

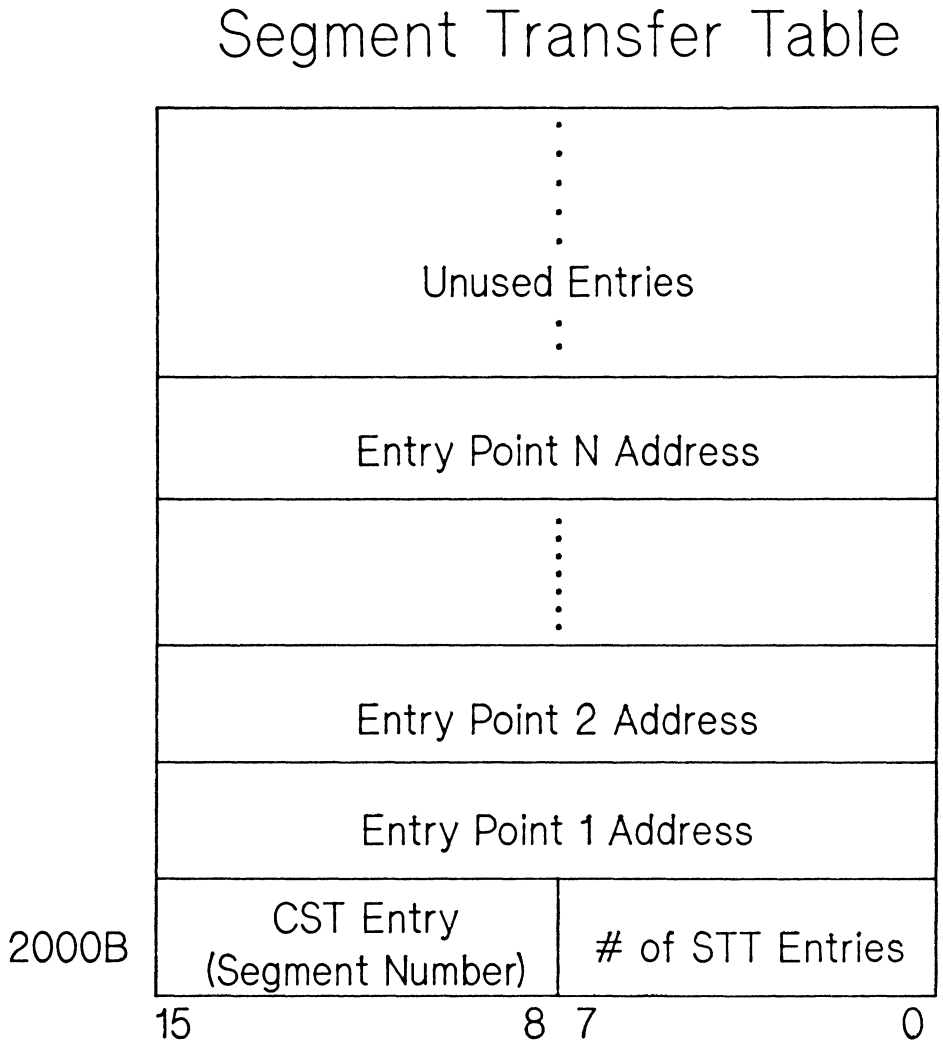

Segment Transfer Table
Unused Entries
Entry Point N Address
Entry Point 2 Address
Entry Point 1 Address
2000B
CST Entry
(Segment Number)
# of STT Entries
15
8
7
0

Figure 5.

PCALR and PCALN call non-CDS code from CDS code. LINK determines which PCAL version to use at load time and patches the correct instruction into the code.

A stack frame is built by the PCAL instruction to contain any dynamic (local) variables used by the called subroutine (figure 2). The Q register points to the first word of the stack frame. The first six words contain the stack marker. This information is used for stack management. STATIC_Q is simply a copy of the Q register and is used by the higher level languages. PREV_Q points to the beginning of the previous stack frame. RETURN_P and RETURN_CST together give the return address and code segment where the address resides. (CST stands for Code Segment Table, which will be discussed next). ARG_COUNT indicates how many arguments were passed. NEXT_Q points to the beginning of the next stack frame. NON_CDS RETURN_P is used as the return address by PCALR and PCALN. Following the stack marker are pointers to the parameters which were passed. These parameter pointers are always resolved for indirects and are not relative to Q. There are ARG_COUNT parameter pointers in the stack frame. Last in the stack frame are the local, or dynamic variables. These variables are defined by the called subroutine. The stack frame cannot exceed 1024 words, which means that the parameter pointers plus local variables cannot exceed 1018 words. FORTRAN can create stack frames greater than 1024 words by performing its own stack management relative to Q. The Z register points to the last word of the stack area and is used to check for stack overflow by the PCAL instruction.

The Code Segment Table (CST) is used to describe each code segment and its current state (figure 3). The CST resides in the code base page from address 0b to 777b. This provides enough table entries for 128 code segments. The number of each code segment is derived from the segment's entry in the CST, from 0 to 127. The CST entry for a code segment shows whether the segment is currently in memory, if it is memory locked, the page location in physical memory, the disc block address of the segment, and the length of the code segment in 128 word blocks.

The Segment Replacement Table (SRT) is used to identify which code block in the code partition contains which code segment (figure 4). RTE-A uses this table to determine which code segment to overlay when a new code segment must be brought into memory, i.e. a segment fault has occurred. Word 0 of the SRT points to the least-most-recently loaded entry in the SRT. This is the next code block that will be overlaid when a new code segment needs to be loaded, and the pointer will then be incremented. The pointer wraps around when the end of the SRT is encountered. The SRT resides in code page 0, beginning at 1000B.

The Segment Transfer Table (STT) contains the addresses of all the entry points into a particular segment (figure 5). Each code segment has its own STT which is located at 2000B in the code segment. Each STT can contain up to 255 entry point addresses. The STT also contains a pointer to the CST entry (segment number) for its code segment, and the number of entry points in the code segment. When a cross segment call (PCALX/PCALV) is executed, the microcode uses the STT in the called segment to find the requested entry point address in the called code segment. The STT is the only place that the entry point addresses exist, the PCALX/PCALV only contains an STT index.

A PCALX/PCALV instruction contains both a CST index and an STT index which represent the called code segment number and entry point into this code segment.

When the instruction is executed, the stack marker in a new stack frame is initialized, the parameter addresses are resolved for indirects and stored in the stack frame. Then, the CST entry for the called code segment is checked to see if the code segment is currently in memory. If it is, the code block containing it is mapped into logical memory. If the code segment is not in memory, an interrupt to trap cell 13B is generated. The CDS fault handler (CDSFH) is then entered. CDSFH uses the SRT to determine which code block to overlay with the requested code segment. The requested code segment is then loaded from disc into physical memory, and mapped into logical memory. The STT index from the PCALX/PCALV is used to find the entry point address in the code segment. At the entry point in the code segment is a word specifying the stack frame size required by the called subroutine. The Z register is then checked for a stack overflow condition, and the stack marker is completed. The called subroutine is then executed. When the subroutine has completed execution, the data in the stack marker is used to find the CST entry to return to, and the return address. The stack frame is then released by changing the Q register to point to the previous stack frame. The caller subroutine is then scheduled for execution at the return point.

When the CDS design is analyzed, the following limiting factors become apparent, some of which can make CDS non-transparent to the user:

* The maximum data segment size is 31K words. If EMA/VMA is used, two pages in the logical map are reserved for VSEG, limiting the data segment size to 29K words. The data segment size is limited because all of the data area must be mapped in all the time, since data references could be made to any portion of the data area at any time.

* Cross segment calls take longer to execute than calls to subroutines contained in the same segment as the caller. Calls to segments not in memory force the code segment to be loaded from disc and overlay the least-most-recently loaded code segment in the code partition.

* The maximum size of a stack frame is 1018 words for data and parameter addresses.

* The maximum size of a code segment is 31k words.

* Code blocks in physical memory are of equal size. If the code segments and code blocks are not of equal size, physical memory will not be fully used.

* Maximum of 255 entry points per code segment.

* Maximum of 255 parameters per entry point.

* Maximum of 128 code segments per program.

The extent to which any of these factors can make a CDS program non-transparent to the programmer depends entirely on how each particular program is designed and what system resources the program utilizes.

Scenarios 1 and 2 discussed previously were designed to illustrate typical problems that can arise when running large CDS programs. In the first scenario, the problem seems to be execution speed. In the second scenario, the problem is shortage of stack area, or space in the data segment in general. Having examined the way in which CDS is implemented, some recommendations can now be made as to how to minimize these types of problems.

In the first scenario, the goal is to reduce the execution time for the program to a more reasonable level. There are several actions that can be taken by the programmer to achieve this.

1. Reduce the number of CDS segment faults which occur during program execution. Each code segment fault delays program execution by a minimum of forty five milliseconds, versus about 45 microseconds to map a code segment already in physical memory into logical memory. To accomplish this, the programmer can:
   * Perform manual segmentation on the program instead of automatic segmentation in LINK.
   * Make code segments as large as possible.
   * Place subroutines which call one another frequently into the same code segment.
   * Memory lock code segments which are called often from other code segments.
   * Make the code partition as large as possible, to allow a maximum number of code segments in physical memory.
   * Group less used subroutines together in the same code segment, causing this segment to be resident on disc most of the time.

   Appendix A provides details on analyzing cross segment calls in a CDS program.

2. Keep VMA page faults to a minimum. As is the case with code segment faults, each VMA page fault generally takes a minimum of forty five milliseconds to be loaded from disc. VMA performs best when accessing data sequentially in VMA. This will generate page faults only when absolutely necessary. If data is accessed randomly in VMA, page faults will probably occur half of the time. VMA performance is the worst when accessing data elements around 1024 pages apart. The programmer can, therefore, arrange the VMA area such that the program will access data elements in the most linear fashion possible.

3. Minimize the amount of data in EMA/VMA. Accessing one element in EMA/VMA takes a minimum of twice as long in FORTRAN compared to referencing a non-EMA/VMA element. If several elements must be accessed by FORTRAN, the access will be several times slower if some reside in EMA/VMA. This is because FORTRAN maps the EMA/VMA pages containing the element into logical memory each time an element is accessed. Appendix B presents some ideas for speeding access to EMA/VMA.

4. If the CDS program uses a large number of small subroutines calling

one another, minimizing cross segment calls can improve performance. With four parameters being passed, it takes more than twice as long to make a cross segment call as a call to the same code segment. This is about 45 microseconds on an A700. If there are less code blocks than code segments for the program, any improvement in this regard will be overpowered by the time lost going to disc to load a new code segment into memory when needed. See Appendix A for information on cross segment call analysis.

5. Routines which take up the most time in a CDS program can be rewritten in MACRO. CDS MACRO is not difficult to understand for the experienced assembler programmer. Appendix C gives information on writing CDS MACRO routines.

6. Take advantage of the VIS routines for large floating point data manipulations. For non-EMA/VMA data, this will improve the program execution time with little penalty. If the data resides in EMA/VMA, the VSEG size in most cases must be larger than 2, thus taking away logical memory pages from the data partition.

The second scenario presents the problem of not having enough memory available in the data segment. This is due to the constraint of 31 pages placed on the data segment size, which is probably the largest limiting factor in the entire CDS scheme. Actions which can be taken by the programmer to maximize efficient usage of the data segment are:

1. Move data arrays to EMA/VMA. Larger arrays should be moved first, since their access will tend to be more linear. Arrays which reside in the static variable area should also be moved first, since this will reduce the size of the entire data segment. This includes arrays specified in local COMMON blocks, arrays which are initialized in FORTRAN 'data' statements, and arrays specified in FORTRAN 'save' statements. Next, arrays which are dynamic variables -- those that reside in the stack area -- should be moved to EMA/VMA. This will provide more stack area when the subroutine referencing these arrays is called, since the array is no longer on the stack. The array will always exist in EMA/VMA in this case, meaning space will always be allocated for it.

2. Use Shareable EMA (SHEMA) instead of system common. This will provide more data space only if system common is greater than two pages. It is not possible to use both EMA/VMA and SHEMA. Access to data in SHEMA will also be at least two times slower than access to system common.

3. Variables declared in a 'save' statement in FORTRAN are placed into the static data area. Reducing the number of these variables will directly increase the amount of data segment available for other uses. Variables not declared in 'save' statements are placed on the stack. It is usually preferable to use dynamic variables such as this, since the stack area is used again by other subroutine's dynamic variables.

4. Constants and variables initialized in 'data' statements are also placed in the static area. If possible, initialize variables during program execution. This will allow the variables to be placed in the stack. An example of this might be a 256 byte array that is initialized to spaces. A 'do' loop during program execution could be used in place of the 'data' statement. Constants and variables common to many routines -- constants such as 1, 2, 3, 256, etc.-- can be placed into a common block and shared between all of the routines. This way only one copy of the constant will exist in the data segment, rather than a copy of the constant for each subroutine which references it.

5. Minimize the amount of non-CDS code which the program references. Non-CDS code takes space directly out of the data segment. If possible, make the code CDS. The code will then be placed into a code segment and mapped in only when called.

As is the case in most programming situations, some of the above recommendations are contradictory in terms of execution time and memory usage. The programmer must make trade-offs based on each particular program's requirements. These include defining the manual segmentation scheme, moving data to EMA/VMA, using SHEMA in place of system common, and static variables versus dynamic variables. Each CDS program must be analyzed individually to ascertain which changes will have the greatest effect on increasing performance.

In summary, as CDS programs become larger, the programmer must begin to take into account the way in which CDS is designed. If this is not done, poor program performance will result when the limitations in CDS are met. Programmers can work around these limitations and improve execution speed by:

* Reducing the number of code segment faults.
* Making code segments as large as possible.
* Making the code partition as large as possible
* Accessing VMA sequentially to reduce overhead.
* Minimizing EMA/VMA data since access time is greater.
* Minimizing cross segment calls.
* Rewriting critical routines in MACRO.
* Using VIS whenever possible.

If the limitations are related to the data segment size, the programmer can:

* Move data arrays to EMA/VMA.
* Use SHEMA instead of system common.
* Avoid the 'save' statement in FORTRAN.
* Share constants/data between routines by placing in a common block.
* Convert non-CDS code to CDS.

Each program must be analyzed for its own particular performance problems. The programmer must then make choices to improve the performance.

Appendix A: Analyzing Cross Segment Calls

Code segmentation determines which portion of the code is mapped into the program's logical address space at any given time. When code from the currently mapped segment calls code in another segment, the called segment must be mapped in. If the called segment is already in memory, a simple re-map of logical addresses occur. If the called segment is not in memory, a segment fault occurs. RTE-A is then called upon to bring in the segment from disc and the segment is then mapped into logical memory. The instructions which accomplish this are PCALX/PCALV and EXIT. If the code from the currently mapped segment calls code in the same segment, no re-map is necessary. The instructions executed in this case are PCALI and EXIT. The following table illustrates the execution time for these calls on an A700 processor:

Call to Same Segment		Call to Different Segment Already in Memory		Call to Different Segment Not Currently in Memory	
PCALI	5.5 us	PCALX	23.0 us	PCALX	23.0 us
1 parameter	1.5 us	1 parameter	1.5 us	1 parameter	1.5 us
indirect	.5 us	indirect	.5 us	indirect	.5 us
EXIT	2.5 us	EXIT	15.0 us	Disc access	30 ms
	9.0 us		40.0 us	EXIT	15.0 us
					45+ ms **

** dependent on disc model and other system activity

As the table shows, a PCALX subroutine call will take a maximum of four times longer than a PCALI subroutine call. If additional parameters are passed, the difference will be smaller. If the called code segment does not reside in memory, then the disc access to retrieve it takes much more time than the actual PCALX execution time.

As discussed previously, there are two alternatives which can be implemented to improve code segmentation performance:

1. Avoid segment faults at all costs. Make frequently used code segments memory locked.

2. Make each code segment as self sufficient as possible. Most of the routines called within one code segment should reside within that code segment. This will eliminate thrashing between segments when cross segment calls occur.

The problem is, how to improve code segmentation performance, since there is no real mechanism available for analyzing the code segmentation.

With this in mind, work was begun on a method to analyze the code segmentation efficiency of a CDS program. The method designed had to have the ability to analyze all code segments for calls between any two segments. Secondly, it must analyze all calls into a particular code segment, including providing the name of the called subroutines in a code segment. A summary of the design criteria

used was (in order of importance):

1. Analyze all calls between all code segments.
2. Analyze all calls into a specific code segment, including analysis on a routine-by-routine basis within the segment.
3. Incur as little overhead as possible in the CDS program being analyzed. This includes both the size of the code required that must be be loaded with the CDS program, and the execution time of the analysis code.
4. Easy to use.
5. Easy to change or modify the scheme.

The assumption was made that most programmers have a good idea of the structure of their code, and that the analysis would only be a tool for the programmer to use for increasing the program's performance, but would not automatically increase performance.

The method designed uses the $TRACE option in the FORTRAN compiler. When this option is turned on, the compiler generates a call to a subroutine called !TFEN on entry, and a call to !TFEX on exit. !TFEN and !TFEX are non-CDS code, and therefore reside in the data segment. They print out the subroutine name on entry and exit. The subroutine name is stored as the first few words of the subroutine in the code segment. The last word of the subroutine's stack frame contains the first word address of the subroutine in the code segment. By replacing !TFEN with another routine of the same name, the new !TFEN can do whatever it wants, whenever a new subroutine is entered!

To analyze cross segment calls between segments, there must be some mechanism to retrieve the caller's segment number and the called segment number. The caller's segment number can be found in the stack frame of the called subroutine. The called segment number, however, does not exist anywhere. By forcing another stack frame to be created, however, the called segment number can be found in the new stack frame (figure 6). The new !TFEN is loaded into a code segment by itself. Each time it is called from a newly entered subroutine, a stack frame is created. !TFEN then calls the real !TFEN (called !TFENR). !TFENR gets the segment numbers from the stack, and passes them to an analyzer program, TRSEG, via class I/O. Execution is then resumed in the called subroutine. TRSEG, being a separate program, takes no resources from the CDS program being analyzed. When the subroutine is finished, !TFEX is called. !TFEX does some cleanup and exits back to the subroutine, which exits back to the caller. When the program is completed, TRSEG is flagged, and produces a printout of the results. Subroutine names cannot be kept during this analysis because it cannot be guaranteed that the called code segment, which contains the name, will not be overlayed by the segment containing !TFEN. The code segment containing !TFEN should be memory locked.

The scheme to analyze all calls to subroutines within a particular segment works in a similar fashion. In this case, !TFEN is loaded into the code segment being analyzed. This guarantees that the code segment containing the subroutine names will not be overlayed (figure 7). !TFEN calls !TFENR. !TFENR finds the subroutine name by getting the address from the stack and retrieving the name from the beginning code of the called subroutine in the code segment. !TFENR also gets the calling code segment. This information is passed to another analyzer program,

Analyze all Segment Calls

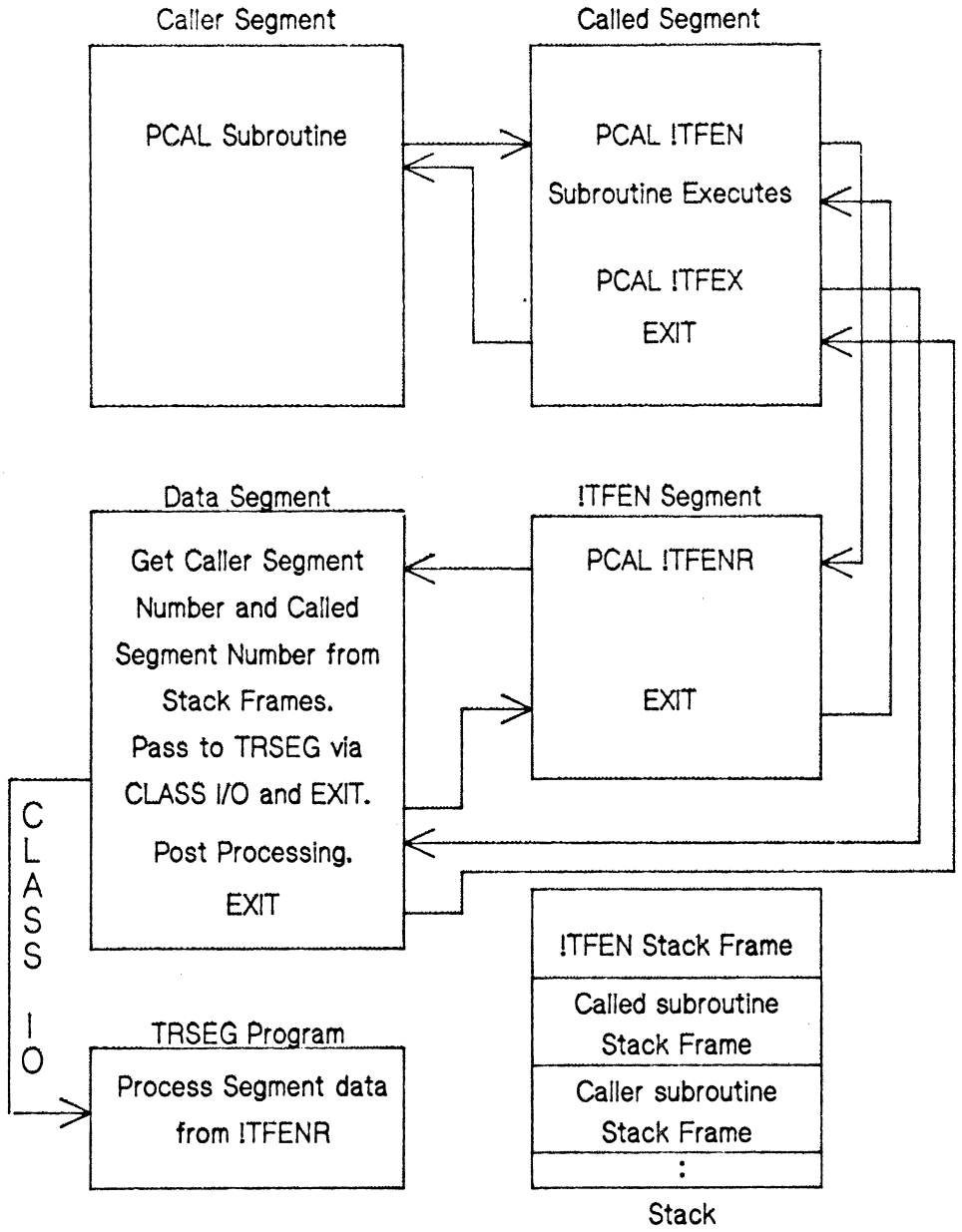


Figure 6.

Analyze Calls into One Segment

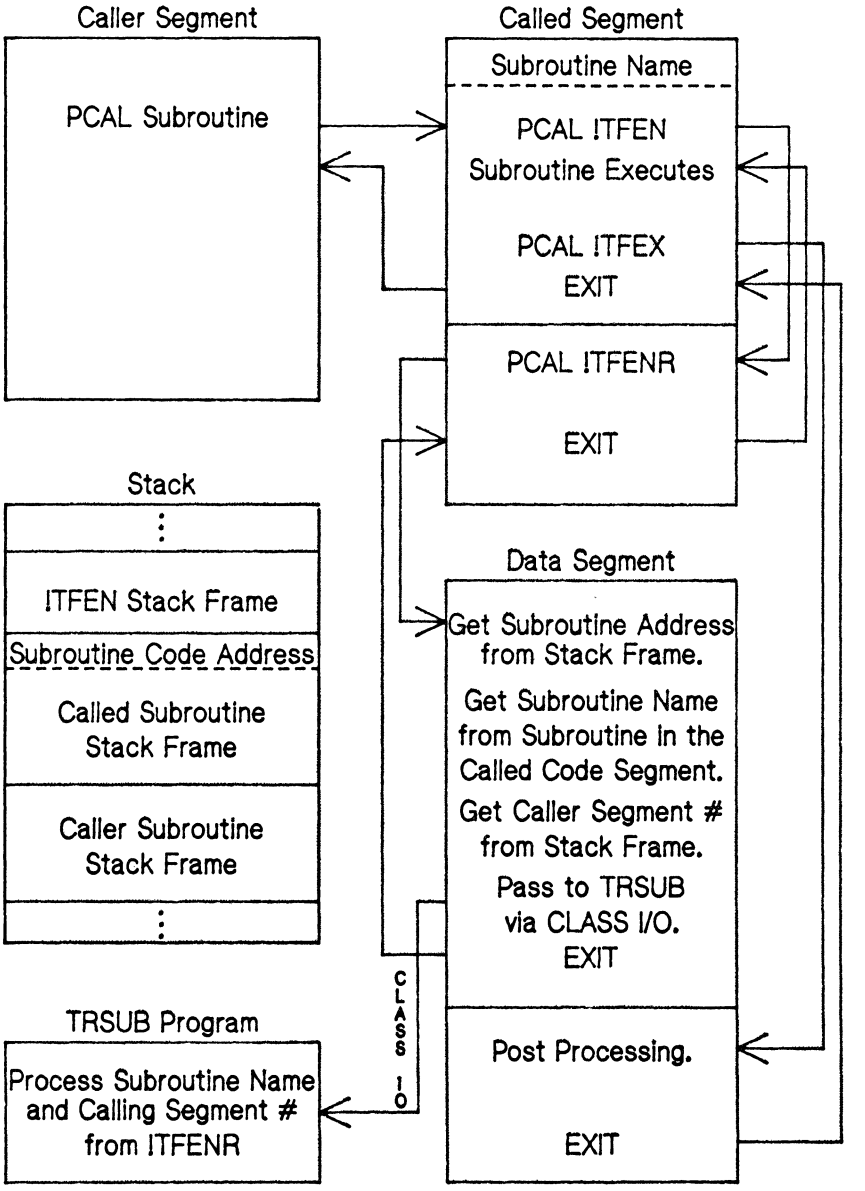


Figure 7.

TRSUB, via Class I/O. Further processing is done in the same way as described above.

In conjunction with this paper, the author is submitting the cross segment call analyzer utilities described in this appendix to the CSL/1000 Contributed Library.

Appendix B: Speeding up EMA/VMA Access

The major problem with moving large amounts of data from the static or dynamic data area to EMA/VMA is a major decrease in speed when accessing the data from FORTRAN. FORTRAN imposes mapping overhead each time an element in EMA/VMA is accessed. This is because FORTRAN can never know where in EMA/VMA the next element is located. It takes FORTRAN less time to remap the page into logical memory than to figure out if the page containing the element is already mapped in. Each EMA/VMA element in FORTRAN has a 32 bit address. The microcode FORTRAN calls uses this address to map the physical page containing the address into logical memory, and returns a 16 bit logical address to FORTRAN. The 16 bit logical address is then used to access the element.

In a CDS program, it is possible that the data in EMA/VMA will contain both large arrays and small arrays, since putting these arrays in EMA/VMA removes them from the data segment. One method which can be used in FORTRAN to speed access to these arrays uses the MMAP subroutine. This subroutine, callable from FORTRAN, allows the user to map any EMA page requested into VSEG. The problem is how to access the EMA data from FORTRAN once it is mapped in. To accomplish this, a $ALIAS directive is used to set up an absolute common block, as shown in the following example:

```
FTN7X,1
$CDS ON
$EMA /DUMMY/
$Alias /mine/=74000B      ! Address of first word in VSEG
      .
      .
      Common /dummy/ Iarray(20000)  ! declare entire EMA size
      Common /mine/  Myarray(10),Mydata(100),Myflag
      .
      My_page=5               ! this routine uses page 5
      Call MMAP(My_page,1)    ! map in 2 pages of data starting
                              !   at my page
      Mydata(50)=1            ! access my data
      Mydata(60)=2            ! no further mapping required!!
```

Using this method, the programmer must know, or assign, on which pages the data resides. This makes the method ideal to use for static data in a subroutine. Each subroutine can have its own page(s) and common block(s) to store static variables. It is easiest to program this method if each common block is a multiple of 1024 words, or one page.

The difficulty with this method is the possibility that VSEG will be remapped if another element in EMA is accessed by the program. In this case the MMAP call must be made again, in order to map VSEG back to the pages desired. The elements in the common block /mine/ can then be accessed again. It is recommended that the programmer use only MMAP calls and no other EMA access, or be very, very careful when allowing FORTRAN to access EMA elements.

Appendix C: Writing CDS MACRO Routines

Writing a MACRO routine using CDS is not difficult. Five important items of which the CDS MACRO programmer must be aware are:

1. The JSB instruction is undefined under CDS. HP supplies a CDS Macro library which includes ENTRY, PCALL, and EXIT for MACRO subroutines. These are documented in the MACRO manual, and make CDS programming much easier.

2. The programmer must define where all instructions and data are to be located. This is done with the RELOC command.
   RELOC CODE - places following instructions into code segment
   RELOC DATA - places following values into data segment
   RELOC LOCAL - places local variables into stack
   RELOC STATIC - places local variables into static data area. Same as FORTRAN 'save'.

3. Since CDS programs use only current page links, the programmer must tell LINK where to place them. This is done using the BREAK command. BREAK commands should follow every JMP instruction, except where the JMP instruction follows an instruction which can skip over the JMP, such as CPA, ISZ, SSA, etc. In general, the BREAK should be placed where it would never be executed, since current page links might be placed there by LINK.

4. There are many new instructions added for CDS. Among these are .CCQA, which will copy the Q register to the A register. This is useful for examining the stack. There are also instructions to modify Q and Z. Be careful!!

5. Some of the base set instructions work slightly differently under CDS or are illegal.

   * Memory Reference Group:
     - All references to base page access Q-relative locations. This is how the stack is accessed.
     - All references to current page access code space and are not Q-relative.
     - JSB is undefined.
     - ISZ, STA, STB cannot reference the current page directly.
   * Extended Instruction Group
     - MBT, CBT, MVW, and CMW are illegal instructions.
     - LAX, LAY, SAX, and SAY resolve base page addresses for Q-relativity first.
   * Vector Instruction Set
     - The interrupt state may not be saved in the code space. This is handled automatically.
   * FFP/Language Instructions
     - .ENTR, .ENTN, .ENTC, and .ENTP may not be used in CDS code.
     - If the word 1 of CFER, DFER, ZFER, or XFER is Q-relative, then all the addresses are Q-relative.
   * Dynamic Mapping Instructions
     - For cross map memory accesses, CDS is turned off.

If the programmer follows these guidelines, after a little practice, CDS MACRO programming will be as easy as non-CDS MACRO programming.

Suppose that a critical subroutine which takes two integer parameters, adds them together, and adds one if the result is odd, needs to be speeded up. Here is an example of what that MACRO routine might look like:

```
macro,q
      HED Example CDS Macro Routine
*
      NAM Fast
      MACLIB $CDSLB::LIBRARIES     * USE RTE-A/VC+ CDS MACROS
      CDS ON                       * TURN ON CDS MODE
*
            RELOC CODE
FAST        ENTRY Value1,Value2,Return_Value
*
            LDA @Value1        * Get first parameter value
            ADA @Value2        * Add second parameter value
            STA Temp           * Save in stack
            SLA                * Odd?
            ADA One            *  Yes, add one
            STA Save           * Save in static data area
            STA B              *   and B for return
            STA @Return_Value  *   and returned parameter
            LDA Temp           * Put sum in A for return
*
*   ALL DONE, EXIT
*
            EXIT               * RETURN
            BREAK
*
*
         RELOC DATA
One      DEC   1
*
         RELOC LOCAL
Temp     BSS   1            * SCRATCH
*
         RELOC STATIC
Save     BSS   1            * Save for later use
*
*
         END
```

1028. MAKING THE MOST OF THE A-SERIES

P.J.Webb
Applied Psychology Unit
Admiralty Research Establishment
Teddington, Middlesex.
United Kingdom

Synopsis

This paper aims to highlight some attributes of the Hewlett Packard HP1000 A-series mini-computer, and to report how its performance has been enhanced by the Applied Psychology Unit (APU). The A-series is used in APU's Human Factors research for the development of real time simulators for future naval system design. Loosely coupled processor networks and advanced communication links have been developed and are now in use at APU. The mini computer has a vital role in the development of various forms of graphics technology and advanced interactive systems to improve the man- machine-interface of future computer based systems for the Royal Navy.

Introduction

The prime function of the Applied Psychology Unit (APU) is to identify and attempt to solve the human factors related problems of future naval computer based systems. This covers a wide range of operational requirements from advanced Command and Control systems to Sonar detection and classification problems and the general use of knowledge based systems in the Royal Navy. The common link throughout this work is the interface between man and machine. The application work is therefore focused on the development of hardware and software tools and techniques to provide improved man-machine-interaction.

The procurement cycle for a military system is generally of the order of three to five years. The rapid advance in computer technology in recent years has often meant that new equipment is obsolete, in technological terms, even before it enters into service. This places a heavy burden on the equipment design team as a decision has to be made at an early stage as to when the design becomes solid, after which no further hardware modifications are allowed for technological improvements. It is therefore essential that as much information as possible is provided to the design team to allow them to maximize the system's performance with the level of technology available to them.

The method employed by APU to investigate and assess the likely performance of these new systems has been to build a simulator of the proposed system utilising advanced forms of interactive display technology and data processing techniques. Royal Naval personnel have then been employed in a series of controlled experiments to assess their performance on the new system.

The system designers problem

Military equipment is generally purpose-built to perform a set range of functions or tasks. This enables the equipment designer to use hardware and software specifically to maximize the systems performance. This is particularly true of signal processing and graphics display systems. The data processed by these

systems is normally provided by various types of environmental monitoring transducer. The human factors researcher and the simulator designer are therefore faced with some major logistical problems. It is essential for the researcher to present the test subject with a system that realistically emulates the real life task: it must feel right, otherwise the performance tests are not truly representative. The simulator designer must therefore provide a system that not only accurately represents the interactive task but also simulates a real world environment, in real time. He must also provide additional processing capacity for on line performance analysis and adequate means for expansion of the experimental task. Thus the simulated task often becomes a great deal more complex than the real system.

Design of any experimental system using advanced forms of technology is, at best, a speculative task. The researcher is often unwilling or indeed unable to provide a detailed specification of the overall system requirement. Thus much of the final design is dependent on the results of a series of pilot studies. This means that the system designer must be able to accommodate many design changes and provide a high degree of flexibility in interfacing to other systems.

A major problem in designing and developing any system using advanced forms of technology is the general availability of the technology required. This is particularly true of interactive graphics systems and demands close co-operation with other governmental, academic and commercial organisations. Utilising new technology in the confines of an existing simulator design is often difficult. The development of special purpose interfaces and low level software tools is time consuming and expensive and is to be avoided where possible.

The adopted solution

The Applied Psychology Unit, like most research organisations, has limited funds and must maximise its use of available resources. With a total staff of only twenty four the development and manufacture of large purpose-built simulator systems is unrealistic. So general purpose commercial processing and display equipment, that can be employed for a wide range of applications, is used where possible. By adopting a modular approach it has been possible to develop a series of building blocks with which to attack most simulation problems. Loosely coupled processor networks have been employed to provide for maximum flexibility and expansion.

The general policy has been to use one manufacturer to allow maximum flexibility in the interfacing and interchanging of processors and peripherals. This has also proved to be cost effective in attracting the more favourable commercial discounts in terms of bulk purchase and volume repair. However no one manufacturer can satisfy our overall research needs so the next best approach has been to split the requirement into an interactive graphics and a data processing task. The interactive graphics task has been the more speculative and equipment has been acquired from several manufacturers dependent on the type of display technology employed. The data processing task has been achieved by using almost exclusively one manufacturer, Hewlett Packard and and one range of equipment the HP1000 mini-computer systems.

Why the HP1000 ?

The Applied Psychology Unit's first experience with Hewlett Packard computers was in 1976 when an HP2100S system was purchased for processing experimental data and simple modelling. Not completely defeated by the HP2100s's 32KWords of memory and the redoubtable RTE-II operating system, we graduated to the HP1000 F-series and RTE-IVB in 1980. The early design studies undertaken on the F-series formed the foundation of our current simulator development programme. The F series was a state-of-the-art computer in its day with some features which still put it a class apart today. The Vector Instruction Set (VIS), Microprogrammability, the open Real Time Executive (RTE) operating system and the Multi Access Controller (MAC) partnership made it easy to customise for real time simulation applications.

The considerable investment in hardware, machine specific application software, the growth of in-house expertise, high system reliability and the overall system compatibility outweighed marginal performance improvements offered by other manufacturers. So with its introduction, in early 1983, we progressed to the HP1000 A-series, initially the A700 and later the A900. The new architecture and increased performance of the A-series overcame many of the problems encountered in the previous systems particularly in terms of Input/Output (I/O). The RTE-A/VC+ operating system, though an improvement on the previous RTE's, falls well short of the ideal. It aptly reflects the view of many Hewlett Packard computer users that the excellent hardware is constrained by mediocre software.

When looking for a processor for this type of application one of the prime requirements is its ability to be interfaced to a wide range of third party equipment. Perhaps Hewlett Packard systems do not immediately spring to mind in this respect, but the openness of the RTE operating system does allow the enthusiast a good deal of scope. The wide range of technically advanced peripherals manufactured by Hewlett Packard which can be supported under RTE also makes the designers task easier. The wide use made of the Hewlett Packard Instrument Bus (HPIB - Hewlett Packard's somewhat individual interpretation of the IEEE-488 standard) has made the interfacing of many third party products a relatively straightforward task. More complex interfacing tasks requiring special purpose driver development have been made easier with RTE-A allowing the user to develop drivers on-line from high level languages employing all the standard debug and test tools. This not only reduces driver development time considerably but also provides a structured, well-documented and more efficient end product.

Processing performance

The processor ideally suited to our applications is the Hewlett Packard HP1000 A-series A900. It is probably the most advanced 16 bit Central Processing Unit (CPU) architecture Hewlett Packard will ever develop. At its introduction in 1983 its performance was superior to most other 16 bit CPUs available at that time and even some contemporary 32 bit CPU systems including the early single processor HP9000 500 series systems. The powerful 3 Million Instructions Per Second (MIPS) CPU, microprogrammability, the capacity to support 24 MBytes of core memory and the high standard of manufacture and reliability at a competitive price makes it APU's favoured solution for the data processing task.

The processing capacity of the A900 coupled with user developed microcode has

provided sufficient performance for most of APU's modelling tasks. However real time operation has often been hampered by the speed of disc I/O when using Virtual Memory Addressing (VMA) and Extended Memory Addressing (EMA) under RTE-A. Dependent on the precise application a variety of options are available. Some improvement can be achieved by providing an additional disc on a separate HPIB interface effectively separating the operating system and data file disc requirements and reducing the disc access bottleneck. Another alternative is to increase the amount of core memory.

The introduction of the 3MByte memory cards for the A900 solved many of our disc bound application problems, at a high financial cost. A system utilising 18 MBytes was soon in service. However under RTE.A a problem was soon apparent. A program requiring very large data areas in EMA, more than 1022 pages, could not directly access the memory. By using VMA, and the "pointers" provided by its use with Pascal, in conjunction with the privilege mode an elegant solution is offered. This not only allows access to data areas far larger than the current A900 memory capacity but adds optional access to the VMA disc area. The elimination of these time consuming disc accesses considerably enhances the performance of the system.

However, for some of our simulation applications the processing power of one A900 has proved to be inadequate for the complete simulation task and it has been necessary to connect more than one A900 in a loosely coupled processor network. Networks of three or four A900's are currently in use for various real time applications. These networks provide an extremely powerful and flexible processing system at a relatively low cost. There are currently fourteen A-series computers linked by various communication networks employed in a wide range of projects by APU.

Communication

With any multi-processor system interprocessor communication is of major importance. Hewlett Packard offer two forms of processor-to- processor communication as standard on the HP1000 A-series. The Distributive System software package, DS1000, operates over a high speed serial link utilising a range of communication protocols. A faster 16 bit parallel link is also provided for high speed data transfer but this is not supported under DS1000 and communication is via direct EXEC calls to the operating system. It has been stated by Hewlett Packard that an IEEE 802.3 (Ethernet) high speed packet switched serial network with Local Area Network (LAN) software, to replace DS1000, will be released soon.

These two types of communication serve two totally different functions. The serial DS1000 network provides a high level network environment, with direct access to many "node" processors that can be thousands of miles apart. It can provide message accounting and rerouting to ensure reception of the signal. However most of the facilities offered, file transfer, shared peripherals and remote program scheduling can be looked upon as domestic tasks. The speed of the serial line, around 28KBytes/sec with HDLC protocol, and the overhead imposed by the control and monitoring software make DS1000 too slow for most real time data transfer applications. With the potential speed improvement offered by the IEEE 802.3 network, 1.25Mbytes/sec, the line arbitration and LAN software overhead is unlikely to provide much more than an order of magnitude speed

improvement on the existing DS1000 system over distances up to 1 Km.

The standard 16 bit parallel link using the 12006A interface card provides a much faster link, over distances up to 6 metres. Benchmark tests carried out at APU have given 714 KBytes/sec on a write and half this speed on a read, which is only half the intended design performance of the card. For our early work this was not a major problem and several loosely coupled processor systems have been developed using this type of link for data transfer. In addition a simple set of utility programs was developed to allow remote scheduling of "CI" over the link, thus allowing the user full use of a remote system from his terminal, providing some of the domestic facilities for software development. This was developed to economise on backplane slot occupation and to eliminate the load on system resources imposed by the DS1000 software.

For our more recent applications the speed of the 16 bit parallel link has proved to be inadequate. An improved link has been developed in conjunction with Hewlett Packard. The new link's benchmark test provides values of 1.75 MBytes/sec for a write with approximately half this speed for a read. The hardware for this link was provided by the specials group at Hewlett Packard Roseville Division. The interface driver and communication software was developed by APU. The driver was written in Pascal with a number of assembler subroutines, and this enables the structure of the driver to be well formed. Testing and debugging was carried out on line without the need for repeated system generation and access to the interface hardware was via the system routines $LIBR and $LIBX. A set of utilities, similar to those for the previous link, was developed for domestic use.

The communication strategy adopted by APU has been to develop application specific processor "clusters", which in turn have been linked by a more general purpose network. Processor-to-processor communication within the "clusters" is not restricted to A900s; high speed links have been developed to connect the A-series to third party graphics processing equipment. Interfacing to non-Hewlett Packard equipment via high speed parallel communication links is not an easy task. Only four communication control lines are provided on Hewlett Packard parallel interface cards; many other manufacturers require six or eight control lines, which invariably means that two interface cards are required in the A-series. This increases backplane slot occupation and necessitates the development of a more complex driver and often requires hardware modification to the interface cards.

Graphics

The nature of the work at APU demands access to the most advanced forms of interactive graphics technology. In recent years raster scan Cathode Ray Tube (CRT) technology has dominated the graphics market place. The demand for greater display resolution and the increase in graphics processing power has caused a departure from the traditional terminal based graphics product. Special purpose raster graphics systems employing high performance graphics processing hardware began to appear in the late 1970s. These systems were generally produced by manufacturers specialising in graphics products, not the major computer companies. In the absence of any suitable system from Hewlett Packard APU chose the best cost/performance graphics processor available at that time.

The Advanced Raster Graphics System (ARGS) was first introduced in 1980 by Sigmex, a United Kingdom based company. An Intel 8086 microprocessor and bit slice technology based system, it supports four Red-Green-Blue (RGB) video processors with 32 dynamically assigned 1024 x 1024 pixel planes. Separate processors sharing a common data bus service the interactive and graphics task. Hosted by both F and A-series HP1000 systems via 16 bit parallel links the HP1000/ARGS partnership has proved very successful over a wide range of applications. The ability to program the local processors within the ARGS has freed the host of much of the interactive graphics task and relieved the I/O load.

Like most high performance frame buffers and graphics processors of this type the ARGS is user programmed at a relatively low level. Order coded display files can be prepared in the host and down loaded to the graphics processor for execution. Data from the host can also be loaded directly into the pixel storage areas. Additionally programs developed in 8086 assembler can be down loaded for local execution by the graphics or interactive processor. This form of software development tends to be specialised and time consuming. Experience with earlier vector refresh graphics systems and the graphics packages that supported them led APU to examine the host resident graphics package approach to software development.

At that time Hewlett Packard offered Graphics/1000 which in its original form was not suitable. Sigmex provided no host based software so after due consideration it was decided to develop a graphics package to support the ARGS. In the early 1980s graphics standards were undergoing a considerable change and APU decided to adopt the Graphics Kernel System (GKS) then a proposed standard from Darmstatd University, Germany. The package took twelve man-months to develop and supports not only the ARGS but most Hewlett Packard graphics terminals. Much of the package needed to be written in assembler so that it is very much Hewlett Packard specific. The performance obtained was better than expected and tests have shown it to be faster than the Graphics/1000 DGL (Device Independent Graphics Library) package when used on Hewlett Packard terminals. Though ideal for many of our more speculative applications, like all general purpose packages a high performance price is paid for adaptability. For applications where performance is paramount a combination of order code, direct pixel dumps and machine-coding is still used. To assist in the development and debugging of the 8086 code in the ARGS a CORAL/8086 cross compiler was purchased for use on the HP1000, so providing a range of host based low level software development tools.

Five ARGS systems are currently in use at APU employed in a wide range of man-machine-interface research applications. These include the presentation of preprocessed sonar data, real time Computer Generated Imagery (CGI) and complex tactical plan development. The technology employed by the ARGS has kept pace with advances in component technology but is now becoming dated. A joint APU and Sigmex team are currently developing a second generation ARGS which is Motorola 68020 based and will employ transputer technology to provide more than an order of magnitude increase in performance.

The introduction of the 12065A video interface card for the A-series was a major addition to the Hewlett Packards process graphics range. The performance is far better than that previously offered on the F series video interface giving higher resolution and multiplane RGB colour on a single interface card. Fully

supported under Graphics/1000-II DGL, application programming is made easy. For high performance applications direct EXEC calls can be used to good effect. Provision of the two RS232 inputs enables the user to connect a wide range of interactive devices directly to the card but with limited support under DGL. A keyboard, mouse and touch sensitive device for use with this interface are currently being developed by APU. Several video cards are in use operating with Touch Sensitive Interactive Displays. The 560 x 455 x 4 bit resolution display is ideal for menu selection and other medium resolution operator workstation applications.

Interactive device development

The investigation and assessment of the various forms of interactive technology is an important area of APU's man-machine-interface research. Hewlett Packard has traditionally provided a wide range of interactive devices which have been directly or indirectly supported on the A-series. However with the emergence of many new forms of interactive technology a variety of new devices have needed to be interfaced to the A-series. This presents a major interfacing problem for each device requires a special purpose interface and host resident device dependent control and data handling software.

The best approach to this interfacing problem is for the device to emulate a standard Hewlett Packard peripheral. The general purpose peripheral chosen was the HP2623 Graphics terminal which allows the use of standard RS232 serial terminal communication protocols and is fully supported for graphics input/output by DGL. A Zilog Z80 microprocessor based emulator was developed which in conjunction with a series of modular device dependent "front ends" enables HP2623 emulation for a wide range of graphics and interactive devices. These include DC Electroluminescent (DCEL) and Plasma graphics displays in conjunction with various forms of touch sensitive technology. A general purpose "console" front end has also been developed for use with a range of experimental operator workstations incorporating switch arrays and a tracker ball. This emulation approach enables host software to be developed with a standard terminal and interactive devices can be interchanged without the need to modify the host software.

On and off display touch sensitive interactive devices are being used increasingly in a wide range of workstation designs. Several forms of touch sensitive technology have been investigated for various interactive tasks by APU. A high resolution infra-red "shadow" system was developed for interacting with graphics displays. This system provided good positional accuracy with high reliability through redundancy. Interfaced to the A-series via the HPIB, four such TSID have been operated on a single bus with other Hewlett Packard HPIB devices.

The area of Direct Voice Input (DVI) is a controversial one which is viewed with some scepticism by the military. A continuous speech recognition system is currently employed at APU, for general experimental use and to assess its performance under "stressful" conditions. A LOGICA LOGOS DVI has been interfaced to the A-series via an RS232 serial and HPIB link. Special purpose control and analysis software is resident in the host with a large library of individual user templates stored on the host disc storage. Feedback of the system's interpretation of the operator's speech input is a vital feature of any DVI implementation. Forms of visual and auditory response are currently under

investigation.

Conclusion

This paper has outlined some of the practical problems that face the simulator designer. The many complex problems involved in systems modelling have not been discussed as they are application specific and to a greater degree machine independent providing the necessary tools are available. The role played by the A-series at APU is a demanding one which often requires the system to operate to the limit of its designed performance.

Some of the good and bad features of the A-series have been highlighted. A range of techniques have been described that enable the user to overcome some of the design problems and enhance the performance of the system. To further enhance the performance of the A-series the user requires some assistance from Hewlett Packard.

If a personal wish list were to be compiled of the improvement that could be made by Hewlett Packard to enhance the A-series a higher standard of system software must be placed at the top of the list. The basic structure of RTE needs to be improved and some of Hewlett Packard's "Historical" concepts need to be revised to take full advantage of modern technology. To enable users to interface more easily to third party hardware the provision of additional control lines would be of major assistance. The range of raster graphics devices needs to be improved; some of the higher performance graphics systems provided on the HP9000 series could be officially supported on the HP1000 range.

With the introduction of the "Spectrum" in late 1985 many of these wishes will no doubt be fulfilled. What further development will be carried out on the A-series in the light of the Spectrum's introduction only Hewlett Packard can say. However the A-series is employed in a wide range of applications, many of which will be in service well into the 1990's, for which the user will wish to make the most of his capital investment.

1029. BUILDING AND USING AN AUTOMATIC TEST DATABASE

Richard Reis
Case Communications, Inc.
2120 Industrial Parkway
Silver Spring, MD 20904

During the past 4 years Case has developed a general purpose 'Universal' test station for functionally testing our products before they are shipped. Test results are printed out onto tickets, which are attached to tested units. They contain the following information:

1. The pass-fail status of a unit - is it ready for shipment.
2. Failed step(s) if applicable.
3. Test parameters - eg. transmit level.

This information is useful in improving quality and lowering cost by helping us to identify and correct design, material, process, and test problems. However collecting the tickets and analyzing the information contained on them could not be done efficiently using manual techniques.

I developed an automated solution to this problem using an HP1000 to 'read', parse, and systematically record information from these test tickets as they are printed at each of the various test stations. Tabular and graphical reports are being produced. Test result reports enable us to focus on and correct the biggest problems. Parameter distributions reports allow us to adjust our design values.

In the future we may expand the system to provide a repair loop and to look at data from the in-circuit component test area.

Introduction:

I will present an overview of the environment that led to the development of the automatic test database and the method of data entry and reporting. I will share some of the tools and techniques used in the development process.

Background and Environment:

Engineers at Case have developed a Universal Test Fixture (UTF) to functionally test products in our factory. The UTF is controlled by a FLUKE 2270 computer through two IEEE 488 busses and an RS 232 asynchronous port. Instrumentation in the UTF takes our products (typically modems) through their paces and evaluates their performance.

Results of these tests are printed onto a test ticket through another RS 232 interface. The printout is up to 40 columns wide by about 30 lines. It contains three types of information:

1. Identification of the unit tested including a bar code serial number, a product number, ...
2. Values for transmit level and other parameters.
3. If the unit under test fails a step the printout contains the step number(s).

Figures 1 and 2 show typical printouts.

Requirement:

We in quality assurance are interested in getting summary information about product test to evaluate the overall level of performance of our product, to set control limits for parameters, and to address and correct the principle causes of failure. The task at hand was one of going from thousands of individual test tickets per week to sensible concise reports. An objective was to make the process as automated as possible.

The product of this effort is periodic and custom summary reports on product system tests. These reports help us to identify design, material, and process problems. Figure 3 is a somewhat whimsical view of a typical test, repair, and reporting system, including the problem identification and correction process. Although this report will focus on the data collection and reporting process, the process of identifying, addressing, and fixing a problem is just as crucial.

Data flow:

The physical configuration of the system is shown in figure 4. The input to the system is the data that is sent to the printer that prints the test tickets. This RS232 data is monitored and sent to the HP1000 using CASE multiplexers. The multiplexers provide error correction and allow us to have only a single cable from the test fixtures. By using low capacitance shielded cable we are able to run the composite RS232 link 500 feet without any signal conditioning equipment.

At the HP1000 site the data is broken out to separate RS232 cables. These lines are read using an HP 12040B multiplexer card in the HP1000, model A600+. Data is then placed into IMAGE data files on a HP7912P disk/tape using FORTRAN programs. The database schema is shown in figure 5.

Software structure:

The software structure is shown in figure 6. A program called SCHED enables all system session programs that are needed for reading, parsing, and storing the data.

A program called LOGT is used to read each of the testers. LOGT was written and linked using code and data separation (CDS) features as a code shareable program. All testers share the code segment of LOGT. Each tester has its own data segment of the LOGT program that reads its records. As each line of the test ticket is read it is placed in a character string array. When the LOGT program detects the end of a test or the beginning of another, it sends the data into system available memory (SAM) using a class write statement. All copies of LOGT use the same class buffer. By running LOGT at a very high priority, I can assure that no copy of LOGT will suspend during a class write. I thus assure that test tickets in SAM will not be interleaved. The information going to the class buffer is an exact copy of the test ticket, with system time and date appended.

RECEIVE reads the class buffer. Each line received is placed into a character

string array. A subroutine, PARSE, uses the character string manipulation features of HP FORTRAN 77 extensively. The intrinsic function, INDEX, is used for locating substrings within the text of the test ticket. Once found, data is copied from the text to character and integer variables. The database subroutine, DBPUT, must be used to load data into IMAGE. DBPUT requires its input data to be in the form of an integer array. I use many equivalence statements to transfer from character strings to the ASCII integer arrays required by the IMAGE database.

A program called TSTR (test report) is used to produce weekly and monthly reports on the data. TSTR uses seven separate tree structures to efficiently store information for test numbers failed, for each parameter measured, and for the test revision number. Recursion, one of the features of HP1000 FORTRAN 77, is used extensively. A subroutine, SET [right or left], creates nodes and records the number of times that that node was visited. The tree is later traversed recursively to create the plots. Printed dot plots show the population at various test limits with a normal distribution overlay. Figure 7 shows a typical weekly report. TSTR also produces another file that is transferred to an HP 150 Touchscreen which is used to produce a bar chart summary.

Future enhancements:

The following features may be added to the system in stages.

1. Test failure description reporting.
2. Active repair loop. The repair technician will retrieve test information from the database, and he will add repair information to the database.
3. Reporting on repair information.
4. 'Expert' repair assistance.
5. Information on the number of units awaiting repair.
6. Application of concepts to in-circuit test area.

A more advanced system is indicated in figure 8.

Tools and methods:

Several special tools were employed to enhance and speed development.

AUTO BOOT is a program which I wrote for the Touchscreen in TURBO Pascal (Borland International). It boots the HP1000 and enters the date and time from the Touchscreen's battery backed up clock automatically (if power is lost and restored) or upon command. Several commercial software packages for the Touchscreen were very useful. Memomaker was even used before the HP1000 was delivered to create the IMAGE schema source. It was and is used for documenting the system including this paper. Diagraph is used for drawing documentation. Picture Perfect is used to produce a bar chart summary of weekly and monthly tests. (Diagraph and Picture Perfect are products of Computer Support Corporation.) An inexpensive but very good spreadsheet, the Planner (Hayden), is a very effective tool for budgets and proposals.

A system session program called WEEK runs every day at 3 A. M. It uses TF to back up the system and runs the past week's report Monday morning.

I have created several miscellaneous programs, subroutines, and functions which I am contributing for the swap tape.

Perhaps the most useful tool was begun during my RTE-A course when I realized that EDIT 1000 lacked a paste buffer. The enhancements are enabled by listing the file, FUNKEYS, to the user's HP terminal. My EDIT screen appears as in figure 9, which also contains the text of FUNKEYS. You can key it in in several minutes. FUNKEYS is also on the swap tape.

Conclusion:

I have spent over one busy year developing the hardware and software described in this report. By presenting this paper, it is my hope that you can use this information to help you and your organization enhance the quality of its products, while improving productivity.

Figure 1
Typical Failed
Test Printout

```
                    Universal System Level Test

Model: R201C                    Control#:00270248
Sta: 2 Rev:1   Date:30-May-85 Time: 00:33
Pn: 905-5147-001   Ser:        Badge:2390

1) AL TEST
     RTS/CTS DELAY (150mS)
2) MANUAL ORIGINATE
     Tx Level PERMISSIVE MODE
     Tx Level=-10.4dBM
     DCD ON AT -43DBM
     DCD OFF AT -48DBM
     ABORT TIMER TEST
3) UUT AUTO ANSWER
     RTS/CTS DELAY (425mS)
***** Error in  3.30 *****
```

Figure 2
Typical Passed
Test Printout

```
                    Universal System Level Test

Model: Intelligent 212 Control#:00269464
Sta: 5 Rev:4   Date:30-May-85 Time: 01:39
Pn: 905-5137-001   Ser:3       Badge:2705

1) Wake up Auto dialer 1200 Baud
   Check Idle Mode RS-232 States
2) 300 Baud Originate
   Pulse Dialing
   Number Expected:&150
   Number Found    :&150
   Establish Line after Dialing
   Bias 48%
   Recieve Space Disconnect
3) 300 Baud Auto Answer
   Send Space Disconnect
4) 1200 Baud Auto Answer
   High Channel;PER
     Tx Level=-10.5dBM
   Reciever Threshold
     -43dBM ON
     -49dBM OFF
5) 1200 Baud Originate
   Tone Dial Test
   Number Expected:&12345
   Number Found    :&12345
   Drops Line after Busy
   Number Expected:&67890
   Number Found    :&67890
   Establish Line after Dialing
   Active RS-232
   Errors    0 (Max    3 Allowable)
   Low Channel;PER
     Tx Level=-10.2dBM
   Programmed Disconnect

   Unit PASSED with  0 Errors
```

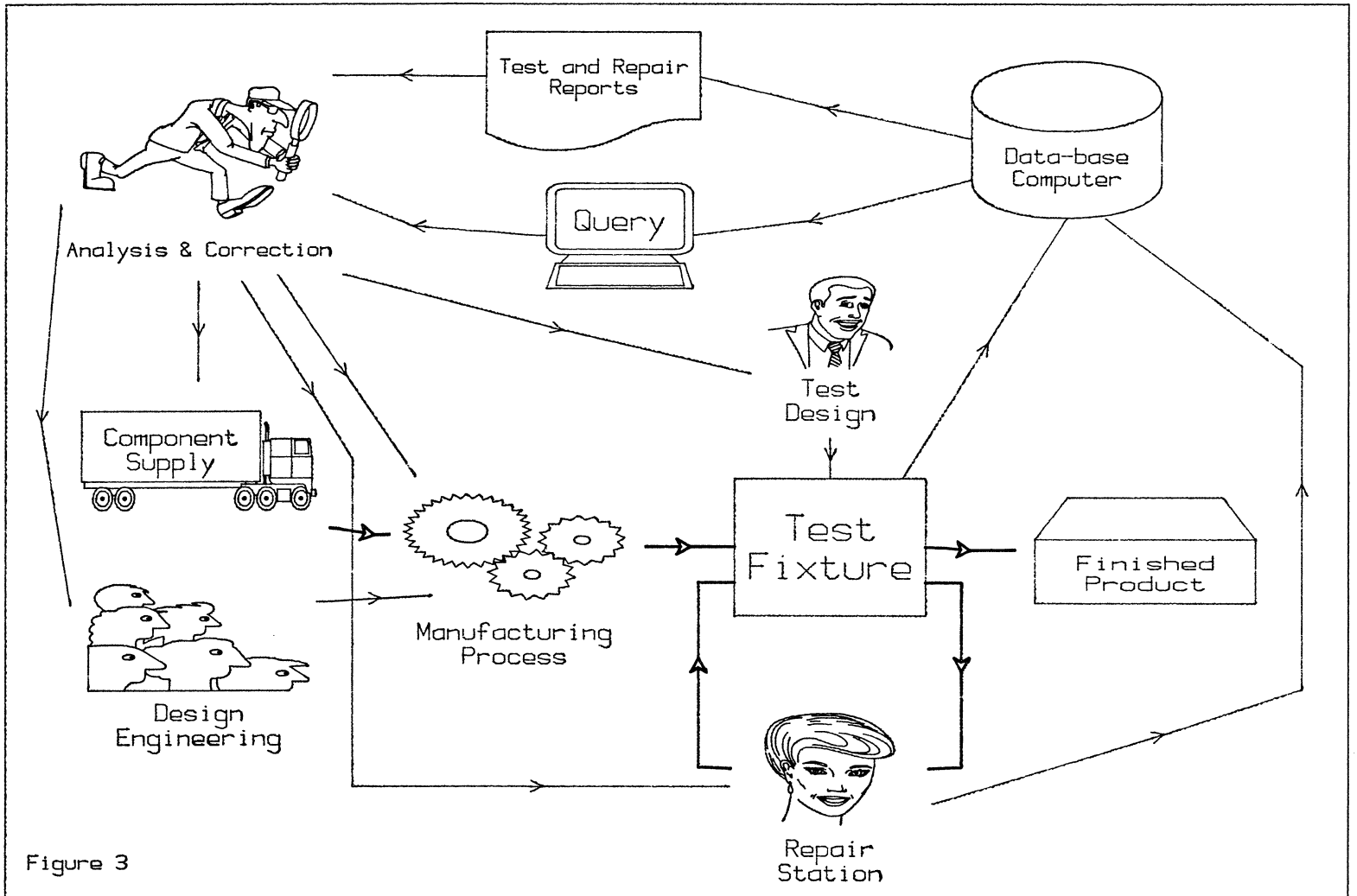


Figure 3

Test, Repair, Analysis, and Correction Process

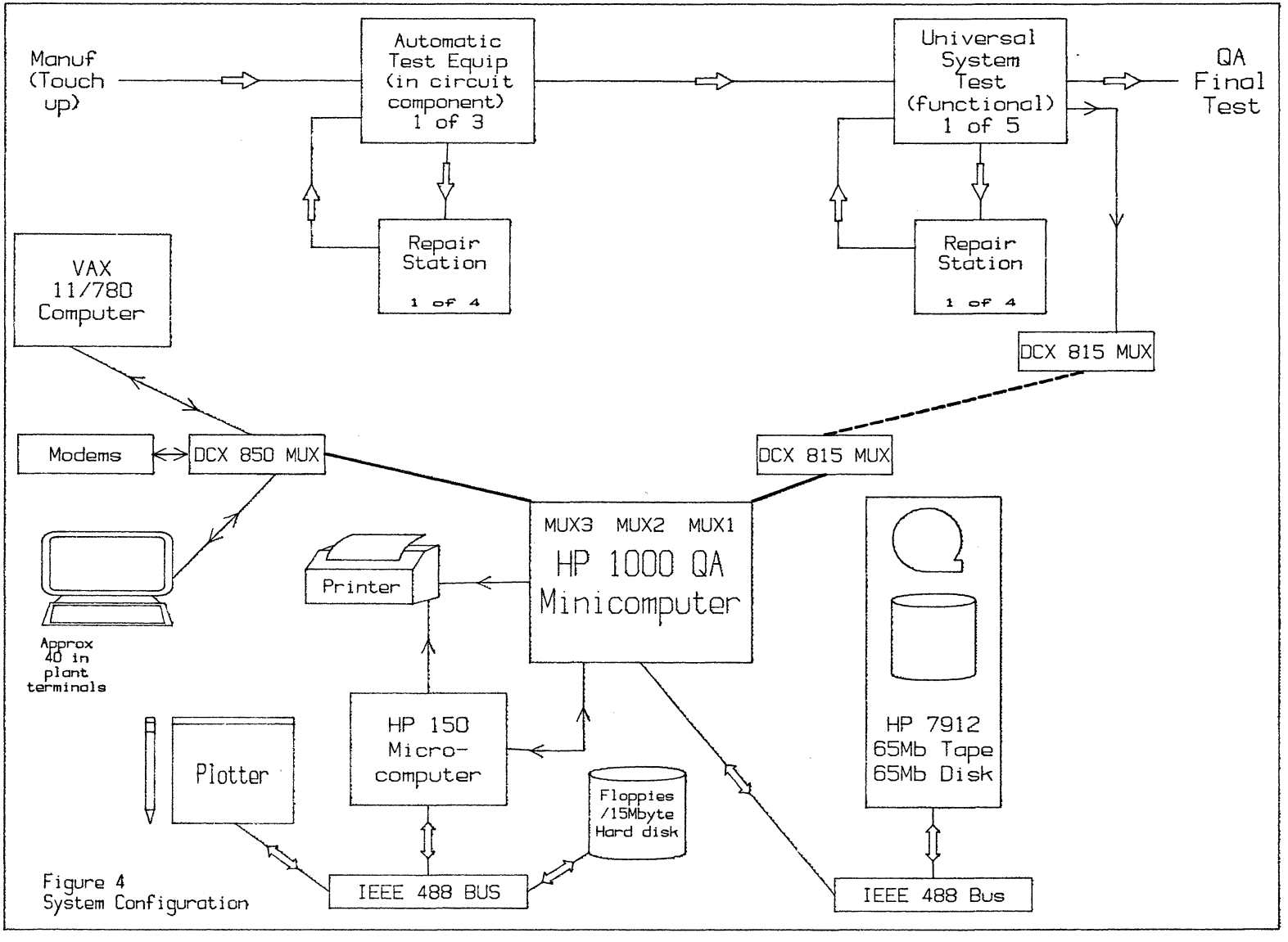


Figure 4
System Configuration

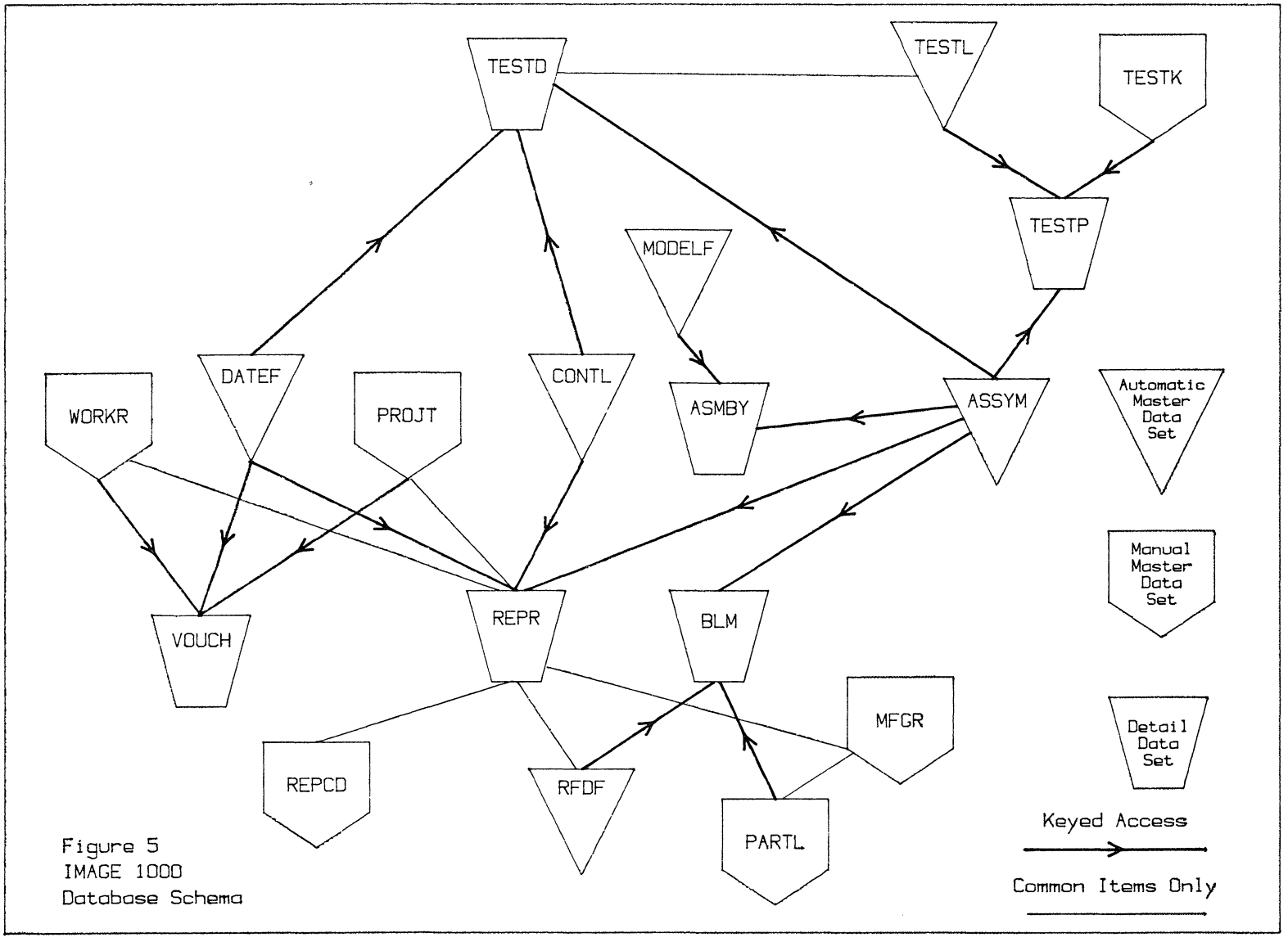


Figure 5
IMAGE 1000
Database Schema

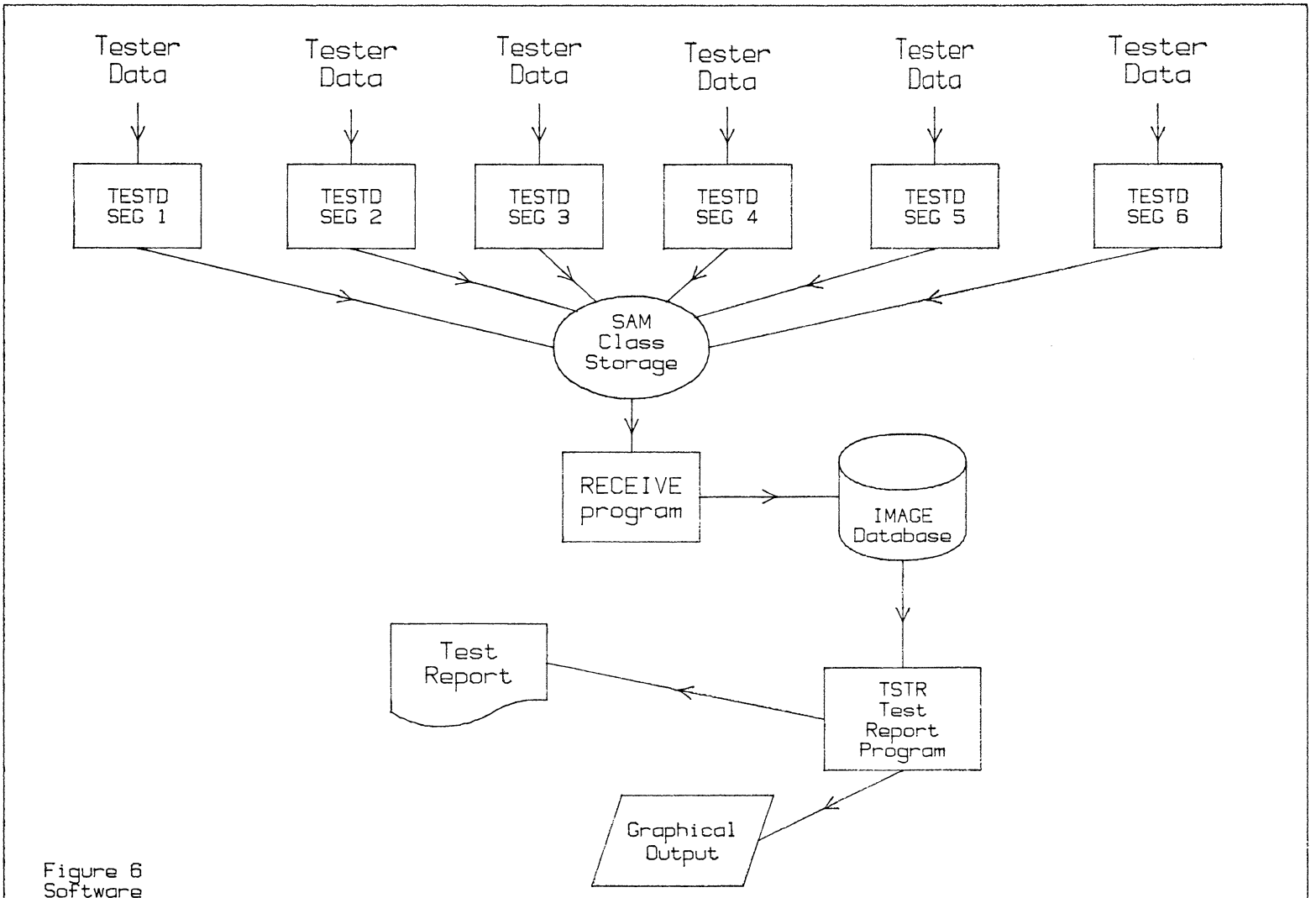


Figure 6
Software
Structure

Report time and date: 1:55 PM WED., 29 MAY , 1985 Page: 2
Model: Intelligen Assy #: 905-5137-001 Test sta: All
Date range of report: 5/19/85 to 5/25/85. New/RGA:New

Population by test revision.
4 100.0% 354XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Population by tests failed. Only first 1 fault(s) shown.

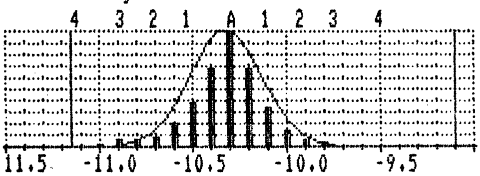
.17	5.4%	2XX
1.07	16.2%	6XXXXXX
1.13	10.8%	4XXXX
2.17	8.1%	3XXX
2.25	13.5%	5XXXXX
4.19	8.1%	3XXX
5.20	21.6%	8XXXXXXXX
misc < 5%	16.%	6XXXXXX

Total number of boards tested: 354
Total number of failures reported: 37

By test failures/board

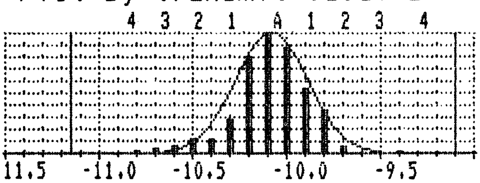
P/F	%Tested	Qnty	
PASS	89.6%	317	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
FAIL	10.5%	37	XXXXXX

Plot by transmit level 1.



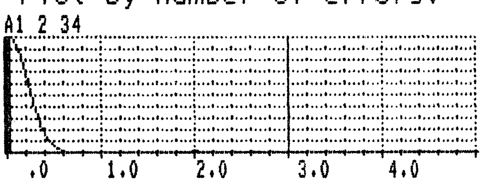
Avg(A): -10.33 Std dev(1..4): .1916
Bars denote population of the data collected.
Line graph shows normal distribution overlay.
Vertical lines denote lower and upper limits.
<--- Scale

Plot by transmit level 2.



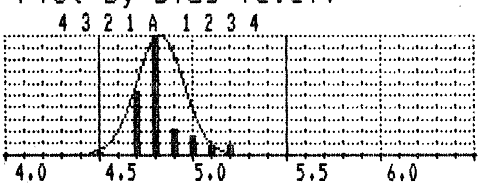
Avg(A): -10.08 Std dev(1..4): .1888
Bars denote population of the data collected.
Line graph shows normal distribution overlay.
Vertical lines denote lower and upper limits.
<--- Scale

Plot by number of errors.



Avg(A): 1.8927E-02 Std dev(1..4): .1936
Bars denote population of the data collected.
Line graph shows normal distribution overlay.
Vertical lines denote lower and upper limits.
<--- Scale

Plot by bias level.



Avg(A): 48.23 Std dev(1..4): 1.250
Bars denote population of the data collected.
Line graph shows normal distribution overlay.
Vertical lines denote lower and upper limits.
<--- Scale * 10^1

Figure 7
Typical weekly report.

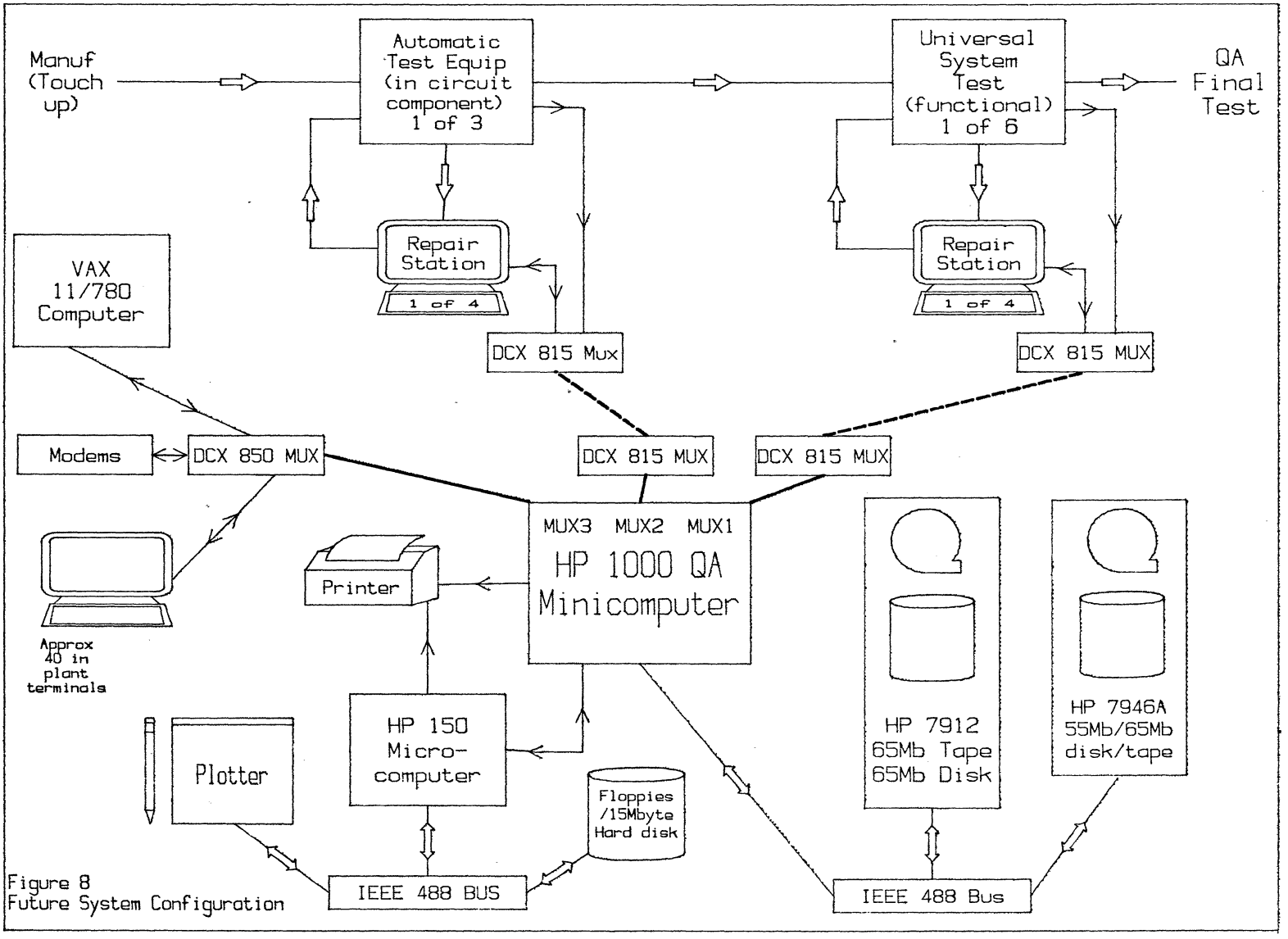
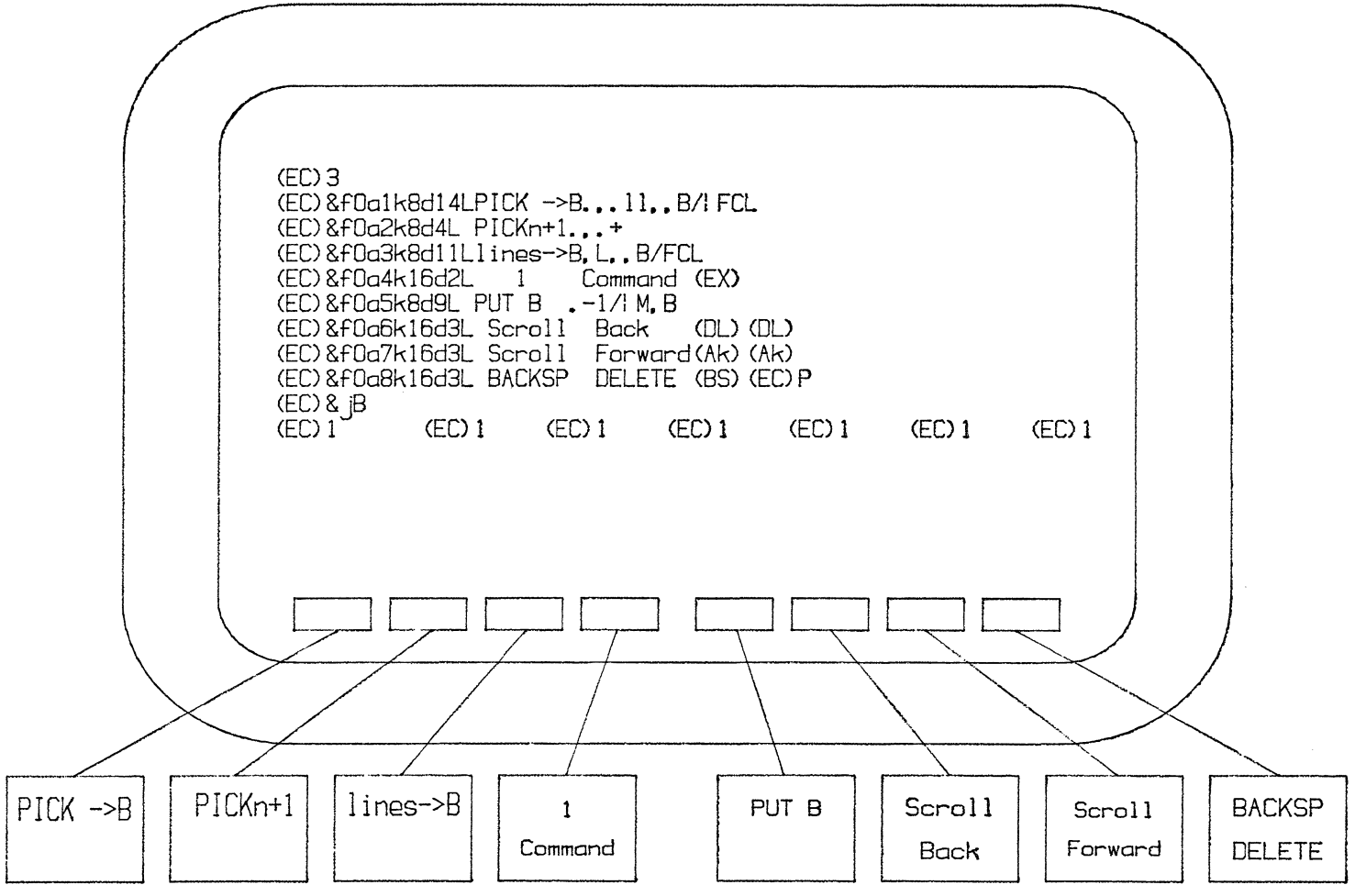


Figure 8
Future System Configuration



Enter text of this file 'FUNKEYS' as shown on screen in screen mode. Press 'system' key, f4(modes), and then f7 (DISPLAY FUNCTNS) and then escape to enter escape(EC) character. Press f7 again to continue text entry. Similiarly enter CNTL-C(EX), CNTL-P(DL), CNTL-F(Ak), and Backspace(BS).

Figure 9
Edit 1000 enhancements

1030.

The Design of Sketch, A General-purpose Graphics Editor

Philip Walden

Hewlett-Packard
Advanced Manufacturing Systems Operation
Data Systems Division
11000 Wolfe Rd., Cupertino, CA 95014

1 INTRODUCTION

The purpose of this paper is to provide the reader with a basic understanding of the design and operation of an interactive graphics editor, called Sketch. The intention is to provide the reader with a case example of how one particular interactive graphics editor was implemented. Hopefully, the concepts covered may be of use in the implementation of other systems with similar requirements.

The reader need not be familiar with the operation of Sketch to derive value from this paper [1]. Many of the concepts may be used in other computer graphics functions.

Several topics will be discussed:

1. Defining the role of an "Editor".
2. Work station independence.
3. Sketch data structures.
4. How the Hewlett-Packard Advanced Graphics Pagkage (AGP) is used.

1.1 A Brief Introduction to Sketch.

The Sketch graphics editor (version 2.0) was first released to the HP 1000 Users Group contributed library in September of 1983. Its primary function was to enable the easy development of overhead projector transparencies (slides) for presentations. However, it also contained features that allowed the user to extend its role into other areas of graphics, such as floor plan layouts, technical illustrations, process displays, and more. All the figures for this paper were created with the Sketch editor. Figures 1 to 4 illustrate some of Sketch's additional capabilities.

Sketch has subsequently been enhanced. The most recent version is 3.3. The major enhancements include: operation on HP 9000 series 500 computers, HP 1000 hierarchical file support, GRAPHICS/1000-II version 2.0 support, a high productivity non-tablet interface, additional functions and performance improvements. This version of Sketch will be the one referenced in this paper [2].

Sketch has its roots in several graphics programs which the author had some hand in the developing. The most noteable predecessor being the slide design program BRUNO. The BRUNO program was released to the contributed library in 1980 [3]. BRUNO provided a truly interactive user environment. The user dealt with relatively few self-explanatory

commands. Feedback was continuously provided with a display of the work in progress. The command set provided the user with a productive method for initially creating the slide and for making changes. In this sense BRUNO was one of the first graphics editors for slide preparation. As a side note, BRUNO was also the design seed for HPDraw (released in 1982), the standard HP product on the HP 3000. (The version of HPDraw available today is still very similar to BRUNO.)

Another program contributing to the present Sketch package was PENNY (1980) [4]. Penny is a schematic drawing package with an especially interesting data structure implementation built upon the HP IMAGE/1000 data base package. It provided a hierarchical scheme for developing drawings. The scheme was well suited to schematics that which usually consist of many instances of the same symbols and shapes. It also represented a different style of interface, since the operator used a digitizer for menu picking, moving, copying and other editing operations.

A third package, called AESOP (1983) must also be considered [5]. It represented a initial trial of lessons learned from BRUNO. It overcame problems in BRUNO's data structures, and created a workable scheme for using the GRAPHICS/1000-II Advanced Graphics Package (AGP).

The above programs contributed to the present day Sketch editor in many ways, both by what they did well and by what they did not do well. As with all examples of evolution, unworkable features or problems in early generations are improved in the next generation.

2 DESIGN OBJECTIVES

The Sketch graphics editor had four main design objectives. As mentioned in the previous section, many of these objectives evolved from problems experienced with previous graphics editors. Failure is the "mother of invention", so to speak.

1. Provide a comprehensive set of functions that are easy to learn and use, yet allow the user to accomplish most any desire.

Many earlier graphics editors provided a large number of functions on the premise that more functions were needed to cover more situations. However, this approach resulted in two problems. First, the user was faced with learning too many different functions and would quite often have to refer to a manual. Secondly, the program itself became large, complex and harder to implement incremental improvements.

The reasoning behind Sketch's array of functions is that fewer is better. Less to learn and easier to use. To maintain the level of capability, the function set must interlock so that combinations of functions can be used to achieve the desired result.

2. Utilize work-station independence as fully as possible without underutilizing the work station features.

Probably the biggest problem with graphics editors prior Sketch, is that they were designed to work with only one or a few work stations and graphics devices. Sometimes it seemed that the software took so long to develop that the target work station always became obsolete shortly after the software was released. Furthermore, often a potential installation site would have the right work station, but not the right plotter, or the reverse.

What was needed was a concerted effort not to "hard code" the editor to any one set of peripherals or machines. Achieving this objective would create an editor with longevity and versatility. The user could use whatever equipment was handy and expect to follow hardware trends as new equipment became available.

3. Eliminate most restrictions associated with drawing size or complexity.

BRUNO had a particular problem which showed up when advanced users pushed the limits of the number of objects in the drawing and when a lot of detailing was required. For example, a drawing could have only so many lines, and so many pieces of text, and so on. Also, as more objects were added drawing speed dropped rapidly. Dense drawings became difficult to edit and users had to save and restore files to recover data space.

Sketch tried to overcome these problems by using a more advanced data structures and capitalizing on AGP capabilities.

4. Use the full power of the Advanced Graphics Package (AGP) in order to reduce overall development time.

Only a limited amount of time was available to develop Sketch. AGP provided a vehicle for designing and implementing a working editor quickly, thus providing more design time to achieve the other design objectives. Also, AGP at the time was in the process of being ported from the HP 1000 to the HP 9000 series 500. Using AGP offered some hope of portability across future HP machines and therefore a longer useful life for Sketch.

3 FUNCTION SET

The function set for Sketch evolved from a prototype set. The driving forces behind the evolution were many: feedback from users, desireable features of similar graphics editors and analogies to similar applications.

In order to keep some semblance of order and cohesiveness to the function set, a de facto decision criteria evolved to decide which new functions were acceptable and which were not. These criteria were the following:

1. Is this feature a basic function of an "editor"?

2. Does this feature meet the needs of the "general graphics" user?

3. Is this function necessary? Can it be decomposed into one or more existing functions? Would a different "mode" of operation on an existing function be more satisfactory?

4. If an option or mode of operation, is it specific to one function or is it applicable to many functions? Modes should have some universal effect on sets of functions in order to keep function complexity minimal.

The original Sketch function set was developed based on analogies to text editors. Text editors have evolved from primitive beasts to rather useful tools over the last several years. So capitalizing on a mature technology would theoretically short-cut some years of graphics editor evolution.

3.1 Functions of an Editor

Earlier graphics editors offered a starting set of graphics edit functions. In the normal evolution of applications, each new generation of editors would build on the previous getting better and better. In order to accelerate this process, many of Sketch's features are based on a similar application technology with a longer lineage, the text editor. The assumption is that the empirical theory of "editing" developed by the successive generations of text editors should be transferrable to graphics editors.

The objective of any editor is to provide its users with a creative and productive tool. The objective is no different with an interactive graphics editor. Therefore it is useful to compare the function of a text editor with that of a graphics editor.

The most powerful text editors basically achieve their productivity

through a combination of a "complete" set of basic editor functions, a generalized syntax and a simple scheme that provides visual feedback of the work at hand. By analogy, a "good" graphics editor should do the same.

3.1.1 Basic Editor Functions

What are the basic set of editor functions? Many text editors divide functions into two major groups: Commands and Text Entry. Text Entry is the operation of keying in text. Commands refer to all other functions. The Commands are of four basic types: Edit functions, Input/Output functions, Viewing functions and Operating modes. There is by analogy direct equivalents in the Sketch graphics editor as shown in table 1.

Function	Text Editor	Graphic Editor
Entry:		lines
	text	text
		arcs
		polygons
Commands:		
Edit	move	move
	copy	clone
	delete	purge
	undo	unpurge
	exchange	edit attributes
		scale/sotate
		stretch
I/O	read	edit drawing
	write	save, overwrite
	merge	merge, insert
	list	plot
Viewing	list	pan
	find	area pick
		set window
		expand window
		reset window
		grid
Modes	global	area pick
	asking	pick okay

Table 1. Text editor functions compared to Sketch

3.1.2 Function Interaction

Most text editors have a flat or networked command processor. Therefore, almost all commands or functions are accessable at the highest level. A "move" can be performed right after a "copy" without traversing a hierarchy of menus. The benefit is productivity, since the user does not waste time jumping from one command to another. A drawback is that the user must learn a large command set and be more cautious about not confusing one command with another.

Sketch also has a large main menu with virtually all editor commands accessable at that level (figure 5). There are still compound functions, such as Load/Save that branch to subfunctions, but these cases occur with the less frequently used commands.

3.2 General Graphics

If Sketch was to be a "general-purpose" graphics editor, the question then becomes "what is general graphics?". Most interactive graphics applications are targeted for some specific application such as drafting, mechanical design or PC board layout. The requirements for these applications can be well defined. A general purpose tool in theory means that the tool could be used in every application. However, one would probably not use the 100-tools-in-one gadget from the local hardware store to fix a watch.

Sketch tries to give the user every flexibility, but it does not specifically address specialized applications. For example, Sketch provides a set of standard media shapes such as 8-1/2 by 11 paper. However, users may still define their own media shapes and sizes. Once a media shape is chosen it can be changed and the drawing does not have to be drawn inside the media area. Some users care little about the actual coordinate system units they work with while others may care a great deal. Again, Sketch provides both options.

3.3 Function Modes

To reduce the size of the command set, modes or options are often provided with the base functions to modify their behavior to suit various requirements. For example, a line draw function could have a continous (connected lines) mode or a single (individual line segments) mode. Otherwise two base functions, continous line and single line, would be required. The advantage of modes, is that the size of the base function set is minimized and is easier for users to learn. The drawback is that each function can have many modes or options specific to it. The results is an overall increase in command syntax complexity, making it harder for the user to learn.

Sketch does use the mode scheme to modify specific base function operations. However, the span of control for each mode typically covers many functions. When a mode is used in several functions it has a universally predictable affect on the operation of the command. The result is a reduced base function set without a substantial increase in complexity.

The Area-Pick mode in Sketch is one example. It modifies the operation of almost all edit operators such as move, clone (copy), scale, rotate, stretch and group. With this mode, these operators work on all objects within a specified area instead of individual objects. The Pick-Okay mode allows the users to confirm object selection before a function is executed. The keyboard mode modifies all commands that use a locate input (user specifies a coordinate value) to read the keyboard instead of using the pointer device. Precise numeric coordinates can be entered in this way.

4 WORK STATION INDEPENDENCE

An interactive graphics program interfaces with the user through a graphics work station. A graphics work station provides the user with a range of graphics devices or tools to achieve a desired result. It is not just the graphics display, but an integration of the the other input and output devices that allows the user to complete a task. Work station devices are often generalized to offer a degree of freedom from specific hardware. Examples of generalized devices are listed in table 2.

Generalized work station device	Function
Display	Displays graphics
Keyboard	Returns alphanumeric strings
Alpha display	Displays alphanumeric strings
Button	Returns a finite range of integers
Valuator	Returns a real value
Locator	Returns a point coordinate (x,y) from a pointing device
Pick	Returns an identifier for an object selected by user with a pointing device.
Stroke	Returns a series of points (curve)

Table 2. Examples of Generalized Work Station Devices

One or more generalized work station devices may be implemented by an actual device. For example, a button function could be implemented by the keyboard associated with a graphics display terminal, or it could be a separate box of buttons plugged in somewhere else.

Since generalized devices can be physically implemented by a variety of hardware, application assumptions on such things as the number of buttons on the button device could limit the user to devices that only match the original assumptions. There may be only eight buttons on the keyboard and sixteen buttons on the box. An interactive graphics program expecting no more than eight different responses might fail when the user presses button 9 on the box.

In an environment where graphics hardware changes rapidly and product lives are brief, interactive graphics programs that presume too much about device capabilities will have equally short lives. Instead of assumptions about device capabilities, generalizations of capabilities

and real-time device decision rules should provide a measure of program adaptability to the changing work station environment. Further, it will increases the breadth of work station the program can support.

Consider the layout menus and work areas of the sketch display (figure 5). Here the basic layout is stored as a set of ratios of screen width and height. The real dimensions inquired from the work station combined with the ratio rules adapt the Sketch display to a specific work station. The menu adapts to the size of the hardware graphics text by the use of a table containing the number of characters for each the family of display devices. If the alphanumeric display is also the same device as the graphics display, then an area the height of two alphanumeric lines is reserved at the top of graphics display area for alphanumeric messages.

Inquiring at run-time for the work station's capabilities and basing interface decision on them, provides a measure of adaptability and maximizes the use of work station hardware. For example, if the work station can perform hardware generated polygon fill, then Sketch will automatically turn on that mode of operation. Otherwise, it remains off. In cases where there are a fixed number of alternatives in a users response, the button device is usually a good choice for user input. However, if the number of choices might exceed the number of buttons, then a back up response device such as the keyboard or pick devices should be used.

In other areas user preferences influence the operation or configuration of Sketch. A user profile file is established to record the users preferred environment. From this file hard copy options, menu colors, line styles and other defaults may be set.

In general, in order to maximize the adaptability of the Sketch editor three guidelines are followed:

1. Never assume a specific display size or aspect ratio.

   Basic screen layout is not a hard coded set of device coordinates. It is stored as a set of ratios. The same is true for any graphics object or artifact that needs to be sized and shaped relative to the display size.

2. Inquire from the work station at run-time as much information as possible about the work station.

   This information is used to make real-time decisions about the operator interface. The AGP inquiry capabilities are very comprehensive and are largely responsible for the adaptability of Sketch.

3. Use decision tables and configuration files to obtain information not available from the work station.

This allows Sketch to adapt to the preferences of the user and the work station environment.

5 DATA STRUCTURES

The data structures used in Sketch evolved over many iterations. Rather than describe its history it may be more useful to describe the data structures used in Sketch relative to two other well known graphics editors: BRUNO and PENNY,

5.1 Overview of Sketch Data Structures

Sketch is implemented in Fortran (77) and is therefore restricted to some extent by Fortran's lack of high level data structures, such as records and pointers. Since Fortran is highly portable along with other reasons, it was chosen as the implementation language.

Sketch drawing data is stored in one large fortran array as illustrated by figure 6a. Pointers and data records are simulated using array indices and variable equivalencing. A heap space and free list is maintained in this array by a set of space allocation and garbage collection routines. In both the HP 1000 and HP 9000 versions, this array resides in virtual memory so that it can be large without requiring large amounts of physical memory.

The drawing data is structured as one or more sets of doubly linked, circular list of object records. The five primary object record types are lines, arcs, text strings, polygons and groups. Group records in turn point to separate lists of object records.

Object records may be of variable length. Text and polygon records take advantage of this feature. The other objects use fixed length records.

Group objects as mentioned previously reference other lists of objects. Thus a group allows a collection of objects to be "grouped" together. The process of "cloning" (copying) a group will only duplicate the group record. Duplication of the group sublist is not required.

File storage and retrieval of drawing data is essentially performed by unformatted Fortran reads and writes directly from the data array to the file system. No formatting is performed in order to minimize file size and preserve graphics data accuracy. A Sketch drawing file is therefore a file copy of the memory array.

5.2 Sketch Compared to Bruno

Bruno is also implemented in Fortran (66). However, its data storage is implemented in multiple arrays as shown in figure 6b. Separate arrays

are used for text records and figure records. Arcs and lines are stored together in another array. Bruno figures are roughly analogous to Sketch groups. Bruno does not have polygon capability.

Arc, line and figure records are of fixed length. The text record is of variable length, but has a maximum of 144 characters including linefeeds.

Bruno's array management routines only manage heap space. The data space of an erased object is marked empty but never re-used. If during an edit the number of characters in a text object is increased over the original allocation, then the original space is discarded and a new space from the heap is allocated.

The figure object record references only the name of a file containing the figure data. Figure data files are a totally different format from Bruno "drawing" files. The more complex figures are in fact created by a separate and totally different digitizing program.

Drawing storage and retrieval is also performed using unformatted, binary reads and writes. However, during storage, each array is sequentially scanned and empty space is not written to the file. Thus, when the file is read back, discarded array space is recovered.

The disadvantages with the Bruno scheme compared with Sketch's are:

1. It is possible for the user to run out of space for one type of object while still having room for others. For example, Bruno applications are typically presentation slides, the text space is usually exceeded first even though plenty of line and figure space is available. Since Sketch data is all in one array, overall data space allocation is more efficient.

2. Frequent editing of text objects resulted in much discarded space. However, this was recovered by somewhat inconveniently saving the drawing and then retrieving it. Sketch text records can be increased dynamically with no additional user effort.

3. Objects were never recorded in temporal order of their creation. Each array is drawn in the order: text, lines and arcs, and finally figures. This is not necessarily bad, but sometimes the order of display may effect the final graphics (especially on crt's).

4. If a figure file was renamed or accidentally purged the Bruno drawing would reference non-existent data. Frequently drawings would deteriorate with age as its figure files "disappeared". In addition, archiving a drawing also required archiving of all the figures the drawing referenced. Sketch loads at one copy of a "figure" into a group sublist. Thus all the original data is saved at that time.

5. Bruno figures are a different format from Bruno drawings. They can

only contain lines and curves (connected lines). Editing capabilities on figures are minimal. Sketch "figures" are the same as drawings and thus have the full complement of edit functions available to the user.

6. Figure drawing speed is slower in Bruno because file reads are required for every instances of a figure.

The advantages of the Bruno scheme over Sketch's are:

1. Drawings are saved in files with the empty or discarded records removed. Sketch drawings are saved in files along with the free (empty) list. Separate utilities provided with Sketch are used to recover all empty space. Although the Sketch garbage collection routine minimizes the necessity for this operation.

2. No space is wasted for pointers and record type tags. Sketch uses five words of each record for object management.

3. Figures do not use much drawing data space since they only reference a file. Sketch groups require one copy of the data to be stored in the data space. Thus a Sketch drawing with many different large groups uses much more data space than the equivalent Bruno drawing.

5.3 Sketch Compared to Penny

The data structures used by Penny is "totally" different. Penny uses IMAGE/1000 for all data storage and data organization.

The data base schema is a recursive and allows a drawing to have a hierarchical structure (figure 6c). The set of Penny objects consists only of lines, text and macros. Macros are again roughly analogous to Bruno figures or Sketch groups. A macro can consist of a set of lines, text and more macros. In this way a hierarchy is formed as each macro points to another [6]. In fact a Penny drawing is simply a macro at the highest level in the hierarchy. The actually data, lines and text, are at the bottom (the leaves of the tree structure) as illustrated in figures 7 and 8.

Another feature is that each record in the data base can be quickly found using the IMAGE key hashing algorithm. One macro can be shared by many drawings. This scheme prevents redundant data and reduces storage requirements. A fall out of this feature is that one database stores all drawings for the Penny user. If done right the database will never duplicate text or line data. This feature is ideal for drawings with many instances of similarly shaped objects, such as a electronic schematic drawing.

Another unique quality of this implementation is that since IMAGE is disc based any aborted Penny session can usually be recovered up to the point of the abort. There was consequently no need for data storage and retrieval since that function is performed "on the fly".

The use of IMAGE also causes some problems. All object records are of a fixed uniform length. Text objects, for example, are restricted to only 32 characters. Object attributes are also limited.

The speed of drawing is limited by the IMAGE throughput. This problem is compounded by the use of multilevel macros. For example the macro in figure 8 which contains 13 lines and 4 text objects requires 25 IMAGE accesses to draw it. To alleviate this problem, some unsupported IMAGE record caching was used in the original Penny.

The disadvantages of the Penny data structures are:

1. The IMAGE file-based data structures are inherently slow on reads. Frequently the user requires a total screen refresh which requires full traversal of the drawing's (macro) tree structure. Performance seriously degrades with each additional user as the disc I/O bandwidth is used up. This is not to say that a file-based data structure is not workable, just that the use of IMAGE incurred too much overhead. Further, a graphics data base is not IMAGE's primary target, thus enhancements to IMAGE since 1979 have not reduced this problem. As an additional note, the first prototype of Sketch used a file-based data structure using RTE type 2 files with record caching. Performance was adequate, but again multiple users degraded performance seriously.

2. The fixed record lengths limit the options for object attributes and object variety. This restriction can be worked around with a somewhat more elegant IMAGE schema. However, data base access procedures would grow in complexity and some length restriction might always exists.

3. The hierarchical nature of the Penny data structures are well suited to applications requiring many instances (occurrences) of identical objects or symbols. Electronic schematics are a good example. However, in applications where most objects are unique, such as presentation graphics or mechanical drafting applications, it is not clear that the hierarchy is indeed useful and may in fact be a burden.

4. All macro references had to be within one database. Thus when the database filled up with drawings, a new data base with a fresh set of "library" macros had to be created.

5. One could not easily transmit "a drawing" to a different user in another database, since all the data for every macro reference also had to be transmitted.

6. Since a macro could be used by many drawings, the problem mentioned earlier of drawing deterioration with age occurred when other users edited, deleted or renamed macros used by other drawings. This problem in addition to the problem highlighted in #4, lead to the practice of creating one copy of a common "library" database for each user. Thus the original intent of the database, to reduce data redundancy, was defeated.

The advantages of the Penny data structures are:

1. The file-based data structure provides an increased degree of session security. Users rarely lose hours work in an editing session due to accidents.

2. The use of a ready-made data base reduces the complexity of the application. Data management, free lists, pointers and more are all handled by the IMAGE data base system.

5.4 Detailed Description of Sketch Data Structures

As briefly described earlier, the Sketch data structures were implemented in one large Fortran array (figure 6a). The drawing data is structured as a series of doubly linked, circular lists of variable length object records, as in figure 9. The data access routines are structured such that almost all the pointer manipulation and garbage collection is handled at the lowest level. The upper level routines do not access the data directly, simplifying their structure for other complex operations such as graphics computations. To describe the Sketch data structures, both the structures and the access routines are presented.

5.4.1 Data Structures

At the lowest level, the data space is simply a large array of double precision real numbers. Character, integer and real data is stored and retrieved from the array by using Fortran equivalencing. Each double real can hold eight characters, four integers or two reals (usually an X,Y pair), or a mixture of all. The choice of a double real as the foundation was primarily made because eight bytes of data could be easily moved with one Fortran assignment statement, as shown below.

```
CHARACTER*8 chars
INTEGER*2   ints (4)
REAL*4      reals (2)
REAL*8      value

EQUIVALENCE (chars, ints, reals, value)
```

```
value = data(index)

data(index) = value
```

At the next level, a package of data, or node, consists of several consecutive double reals (figure 10). Each node has a one double real header containing four integer pointers and counts. To address data in a node, the array index of the node, the data index within the node and the data position within the double real is needed.

Usually the header contains two pointers for the doubly linked list (next and previous nodes), a node length (in double reals), and a next extent pointer. An object record consists of one or more nodes. Objects such as text or polygons can grow with over time as more data is added to them. Thus nodes can have extents referenced by the extent pointer. Initially, the extent pointer value is null (zero). When the record is about to overflow the node, an extent node with the another header is created automatically. The extent pointer on the original node then references the extent node. Nodes can have any number of extents.

Nodes are created out of one of two places, the free list or the heap space (figure 11). The free list is a circular doubly linked list of discarded array space created by purging (erasing) an object. The heap is the remaining virgin space at the upper limits of the data array. When a node is to be created, the free list is scanned first to find a space large enough for the node. If space is found the node is created there and any remaining space is returned to the free list. If no space large enough is found then the lowest part of the heap is used. When space is returned to the system, the free list is again scanned. First to join any unused adjacent spaces and then to insert it in the free list in physical (index) order. If the returning space is adjacent the heap, it is returned to the heap. Thus, as objects are purged the heap space can also be restored. As new objects are created, they will tend to re-use space closest to the start of the array.

Object records overlay one or more nodes and can be formed into sequential lists of records. A Sketch drawing consists of one or more object lists, as shown in figure 12. The main level list grows as each object is added to the drawing. The object record is added to the end of the list. Secondary lists are formed for groups. A group is formed by adding or moving objects to the secondary list. The secondary list is referenced from one or more group records that contain transformation parameters for the group. The transformation parameters are used to translate, scale, rotate and stretch that instance of the group on the display. It would have been possible to create a hierarchical with this structure, but limitations with AGP do not allow more groups records in the secondary list. Therefore, third level or higher lists do not exist in a Sketch drawing.

Another temporary list is also maintained with the unpurge list. When objects are purged, they are first moved to the unpurge list where they

are kept until the next purge operation. At that time they are really erased. To unpurge an object, the record or records last purged are simply moved back to the main list.

There are many types of records in a Sketch drawing. Obviously there are the actual object records: line, arc, text, polygon and group. Then there are drawing data records: main list root record, group sublist root records, color table record and the polygon style table record. A record type is identified with a tag field as the first integer of each record (after the header).

5.4.2 Data Access Routines

The Sketch data access routines are stratified into levels in order to reduce the complexity of accessing a dynamically linked structure and yet maintain flexibility in their use.

There are three levels of access routines. The first level is used to directly address the array and pack or unpack data into the double real format. With the HP 1000 version it also serves to isolate the rest of the program from the EMA (extended memory area) addressing peculiarities.

The second level provides node addressing. The calling routine only needs the node pointer (index) value and the relative data index in the node. This level also returns the header pointer data.

The third level routines provide record level addressing. At this level, the calling routine only provides the record pointer (index) value and the relative index into the record. The third level routines automatically account for node extents and will allocate new node extents automatically when needed. Thus the record appears to the calling routines as being one large node having no limit in length and expanding automatically as needed. An object record is therefore much like a memory based mini-file.

Higher level routines exist to access object records in a specific manner. For example, both variable length objects, text and polygons, have a set of file-like access routines:

```
open_text_record          open_polygon_record
   get_text_line            get_num_vertices
   write_text_line          get_next_vertex
close_textrecord          close_polygon_record
```

6 HOW AGP IS USED

The Sketch editor relies heavily on HP's Advance Graphics Package (AGP) [7]. The Sketch software can almost be considered as an interactive shell over AGP. In fact, almost every subroutine provided by AGP is used within Sketch. Besides the fundamental mechanics of the graphics itself, there are several features of AGP that make it particularly useful for the Sketch editor. Those people who have used both AGP and Sketch will probably recognize many similar terms and functions. For example, the Sketch color and polygon style tables are simply manifestations of the equivalent AGP tables. However, there are other less obvious uses of AGP as explained below.

6.1 Work Stations

The AGP work station concept is heavily used in Sketch. Essentially, a graphics work station consists of several basic graphics devices integrated into a work station.

AGP provides two output devices, the display and alpha device, and five input devices, the locator, pick, keyboard, button and valuator devices. Sketch makes use of of all these except the valuator. These AGP devices can be physically implemented as one or more real devices that will act as an integrated work station, as described previously.

As an example, an entire work station could be implemented by one HP 2627A color terminal. The 2627 through AGP provides all the six device functions expected by Sketch. Alternatively, a work station could be implemented by an HP 12065A video card with monitor, an HP 9111A graphics tablet and an HP terminal. The video card and monitor provide the display, the tablet provides the locator, button and pick, and the terminal provides the keyboard and alpha display. The AGP work station integrates their operation such that the same Sketch program operates in a functionally similar manner, regardless of the actual hardware configuration.

To achieve a measure of adaptability over a wide variety of work stations, Sketch makes extensive use of the AGP work station inquiry routine, JIWS [8] [9]. This information is used to make real-time decisions about the user interface. Table 3 lists the type of information inquired and its use.

Information inquired	Use
Display aspect ratio	Anything related to the screen.
Display device name	Miscellaneous tests not covered by AGP: e.g. hardware highlighting

Locator device name	Is locator same device as display: Can locate limits be changed
Button device name	Is button same device as display: Is use of buttons easy
Alpha device name	Is alpha same device as display: Do two lines of the display need to be reserved for the message area
Number of alpha text lines	How much space for the message area
Number of buttons	Limit value
Hardware text minimum width	How many chars on the menu names
Number of line styles supported	Limit value
Number of polygon styles	Limit value
Number of colors	Limit value
Retroactive polygon style changes	Redraw needed after change
Hardware polygon fill	Sets state of fast polygon mode
Retroactive color table changes	Redraw needed after change
Background color modification	Capability limit
Color table modification	Capability limit

Table 3. Work Station information required by Sketch

Generally, experience over both the HP 1000 and HP 9000 versions of Sketch have indicated that this scheme of work station independence does work with most graphics devices to date. Few exceptions were encountered where more than the information supplied by AGP was needed to support the work station. The primary areas where more information was needed (and thus coded into Sketch) dealt with the interaction between alpha and graphics planes on the same display device. The HP 2627A was the most difficult. Background color and the alpha plane color did not mix well and extra routines were needed to modify the alpha plane color when the graphics background color changed. The other notable exception was for the 12065A video card for the HP 1000 A-series. Here the device performs hardware highlighting (blinking), but AGP does not have a specific inquiry to return that information.

In the future when window manager interfaces become more prevalent, graphics displays could vary their aspect ratios and other characteristics "on the fly". In this case a graphics editor would have to trap such changes and make appropriate accommodations.

6.2 Segments and Picking

AGP provides the object picking facility for Sketch. A pick function returns the identity of the object selected by the operator. This is a fundamental operation required of any graphics editor.

To make use of the AGP pick function, individual objects are recorded in the AGP segmented display area (SDA) as AGP segments. When an AGP pick function is invoked, AGP scans its SDA and returns the segment identity (id) of the nearest segment (object) to the pick coordinates.

AGP segment id's are simply integers. Sketch equates the segment id directly to the object pointer, another integer. Since AGP uses single precision integers, the Sketch data array size cannot exceed 65,534 bytes without some other method of equating AGP id's to object pointers.

Object erasure is another facility provided by AGP through use of segments. To erase an object from the screen a call is made to the AGP segment purge routine with the object pointer used for the segment id. The blinking of lines and arcs is performed by the AGP highlight function. The active menu function boxes are again the AGP segment visibility functions. An interesting use of segment visibility is required for displaying the color table or polygon tables. The entire screen display is made invisible by one AGP call (JVSAL) [10]. After the table function is completed, the entire screen is returned to visibility with one call, restoring the original screen.

6.3 Modelling Matrix

The AGP modelling matrix (JDMOD, JCMOD) provides a useful method of transforming objects [10]. In the case of Sketch, the modelling matrix calls are used to transform groups. A group is simply a fixed list of objects referenced by a group record. The group record contains the values for the 4x4 modelling matrix providing translation, scaling, rotation and stretching of the group list.

The advantage of this method is that the group list data need never be modified to perform a graphics operation. Only the modelling matrix is modified to provide the operation. Since a group list can be referenced by many group object records, it is also a convenient for creating similar objects with one set of data.

A drawback to using the AGP modelling matrix is that it cannot be altered more than once per AGP segment. Therefore, a group list can never contain another group record. If it did, AGP would report an error when the group record in the group list caused the modelling matrix to change for a second time. This is the primary reason why Sketch does not provide a hierarchical drawing structure, even though the underlying data structures could accommodate this capability.

6.4 Difficulties using AGP

Although AGP has proven to be an extremely useful tool in the development of Sketch, it is not perfect. There are many areas where changes or enhancements would result in its easier application. The following sections outline areas where AGP could be improved.

6.4.1 The Duplicate Data Base Problem.

There are two data structures, or databases, within Sketch. One is controlled by the Sketch program and the other resides as the SDA within the AGP work station program. As with any duplicate data base situation, there are problems with keeping the duplicated data synchronized.

Sketch's data structures store graphics and higher level information such as order, groups, arcs, and more. The SDA stores primarily basic graphics data grouped into segments for picking and other operations. The AGP SDA is basically a write-only data base. It does not provide read capability on for its graphics data. Thus both Sketch and AGP duplicate the graphics data in their respective data bases. If the SDA graphics data could be read back by the application, then all graphics data could be stored in the SDA and no duplication would be necessary.

6.4.2 Multiple Segment Operations

Sketch makes use of AGP segments stored in the SDA. AGP provides a set of segment manipulation functions that affect segment visibility, detectability, and highlighting. However, the control over these functions is limited to the point that often they cannot be used because of negative side-effects, even though they offer attractive savings in application code and development time.

6.4.2.1 Limited segment selection

Originally Sketch maintained the basic menu and screen layout as a fixed set of AGP segments. To clear the work area and refresh the display, all object segments had to be purged first. The menu segments were left alone. Thus, the menu and screen layout was drawn only once at program start-up. The menu was automatically maintained by AGP!

However, the segment purge routines only allow erasure of one segment at a time (JPURG) or the entire SDA (JCLR), including the menu. Therefore, Sketch had to scan a list it maintained of visible object segments to provide the selective purge. This scheme was eventually dropped because the AGP purge of a large number of individual segments took too long. Currently, Sketch purges all segments and redraws the menu each time. Thus bypassing what could have been a useful feature in AGP.

In general, more comprehensive segment selection scheme could be

provided. Examples are:

1. Selection by area. All segments within a specified area are selected for the operation.

2. Selection by range. All segments with identifiers greater or less than a certain value are selected for the operation.

3. Selection by set. Given a set of segment identifiers, all segments in the set are selected for the operation.

6.4.2.2 Display and SDA synchronization

Within the AGP work station program there is another synchronization problem where the contents of the SDA must be the same as the graphics displayed on the screen.

AGP is somewhat overprotective in this area. If the application attempts to modify the SDA without changing the graphics on the display, eventually AGP will automatically force a new-frame-action. The new-frame-action clears the graphics display and redraws the display from the contents of the SDA, thus ensuring synchronization. This phenomena occurs primarily when Sketch knows a segment operation will not affect the graphics of the display, but requires restructuring of the SDA. For example, merging separate objects/segments together as one segment (grouping).

Originally in this case, the work station was turned off via a JWOFF, and the objects/segments involved were purged and redrawn under in one segment. The effect was to prevent object erasure and redrawing on the graphics display. However, AGP would later cause a total screen erasure and refresh when another unrelated segment operation was performed. This side-effect was annoying and currently Sketch puts up with the object erasure and redrawing.

In this case AGP could be changed to 'trust' the application when it does things that might cause display/SDA inconsistency.

7 SUMMARY AND CONCLUSIONS

The previous sections briefly outlined several areas about the implementation of the Sketch graphics editor. These areas represent the most important lessons discovered during the design and implementation of Sketch. Lessons which can be leveraged into the implementation of other computer graphics programs

The areas covered were:

1. The function of a graphics "editor". Modelling a relatively "new" application after "tried and true" applications in different but similar environment thus reducing the evolution time of the "new" application.

2. The need for work station independence and basic rules for achieving it.

3. The data structures used in the Sketch editor, their implementation and their access methods.

4. The leverage provided by the Hewlett-Packard Advanced Graphics Package (AGP). Both the uses and difficulties with it were covered.

References

1. Walden, Philip: The Sketch System Operator's Guide and Reference Manual, Sketch - General Purpose Graphics Editor Version 2.0. CSL/1000 Program Library of Users' Software for HP 1000 Systems, Interex, The International Association of Hewlett-Packard Computer Users, 2570 El Camino West, Mountain View, California, 94040, September, 1983.

2. Walden, Philip: The Sketch System Operator's Guide and Reference Manual, Sketch - General Purpose Graphics Editor Version 3.3. Hewlett-Packard, 11000 Wolfe Road, Cupertino, California, 95014, January, 1985.

3. Long, Jim and Walden, Philip: BRUNO. CSL/1000 Program Library of Users' Software for HP 1000 Systems, Interex, The International Association of Hewlett-Packard Computer Users, 2570 El Camino West, Mountain View, California, 94040, January 1, 1983.

4. Walden, Philip: Penny Programming and Reference Manual. CSL/1000 Program Library of Users' Software for HP 1000 Systems, Interex, The International Association of Hewlett-Packard Computer Users, 2570 El Camino West, Mountain View, California, 94040, September, 1979.

5. Key, Scott: AESOP - A Slide Development Package. CSL/1000 Program Library of Users' Software for HP 1000 Systems, Interex, The International Association of Hewlett-Packard Computer Users, 2570 El Camino West, Mountain View, California, 94040, June, 1983.

6. Walden, Philip: PENNY - Computer Aided Drawing on the HP 1000. Communicator/1000, Hewlett-Packard, Data Systems Division, 11000 Wolfe Road, Cupertino, California, 95014, Vol. III, No. 6, 1979, pp 38 - 47.

7. Advanced Graphics Package User Guide. Hewlett-Packard, 11000 Wolfe Road, Cupertino, California, 95014, Part No. 97085-9000, June 1983.

8. Advanced Graphics Package Version 2.0 Supplement for HP 1000 Systems. Hewlett-Packard, 19420 Homestead Road, Cupertino, California, 95014, Part No. 92862-90001, pp III-15 to III-23, May 1984.

9. Advanced Graphics Package Version 2.0 Supplement for HP 1000 Systems. Hewlett-Packard, 11000 Wolfe Road, Cupertino, California, 95014, Part No. 97085-90001, pp III-14 to III-22, June 1983.

10. Advanced Graphics Package Reference Manual. Hewlett-Packard, 11000 Wolfe Road, Cupertino, California, 95014, Part No. 97085-90005, June 1983.

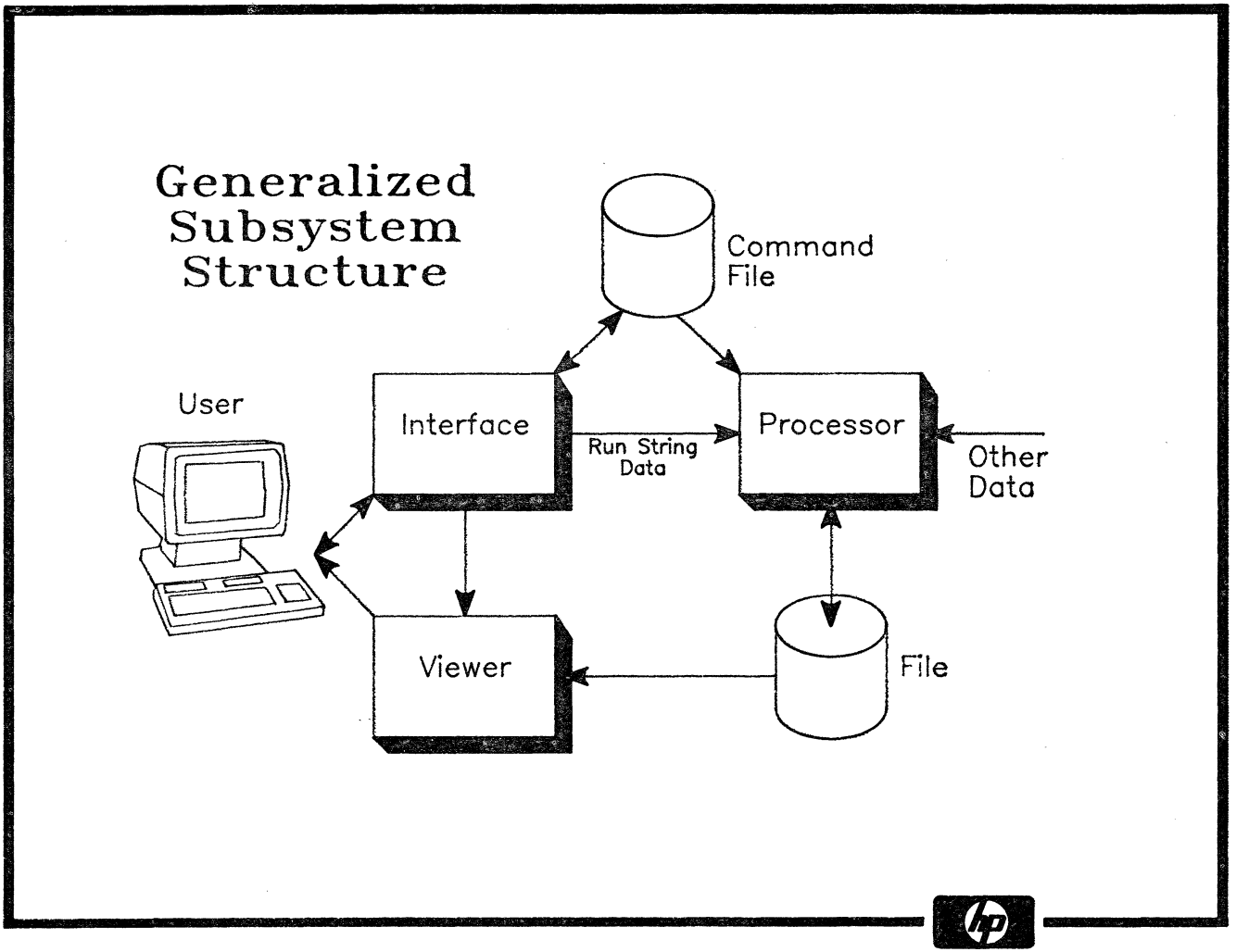
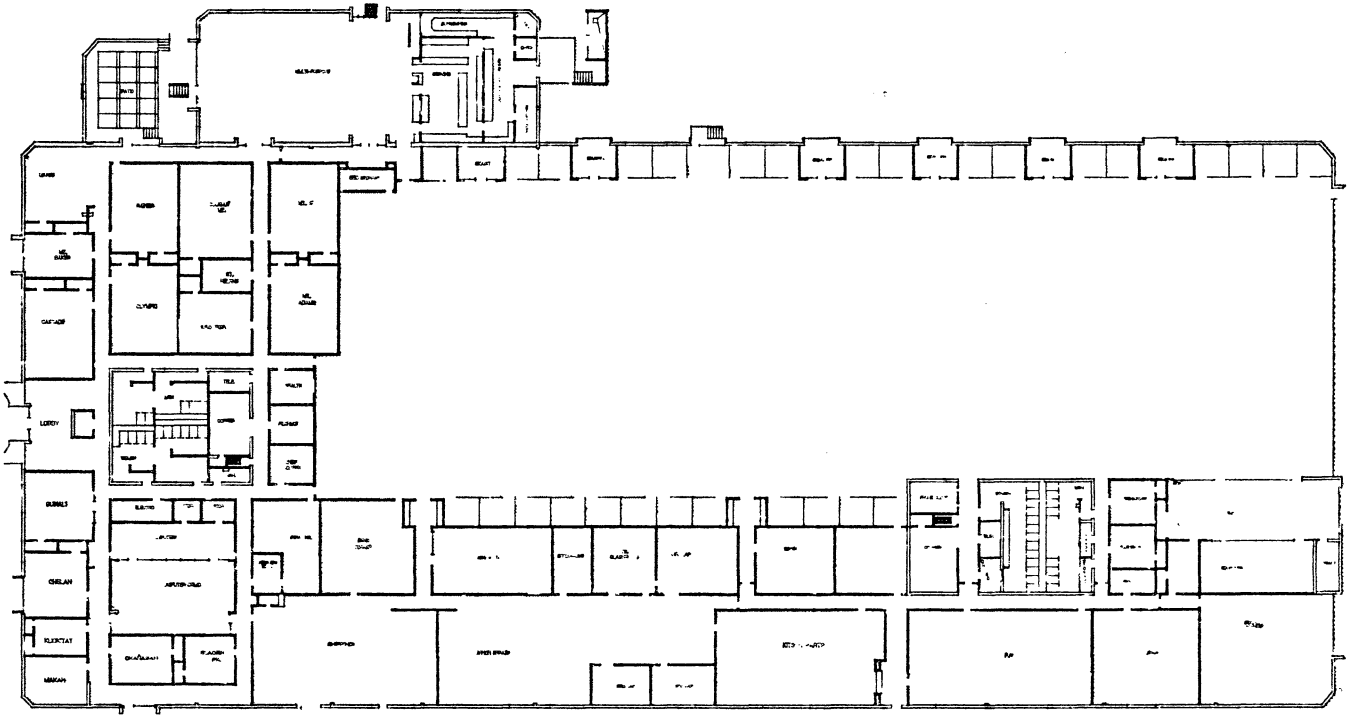


Figure 1. Overhead Transperency created with Sketch.



Source: Scott Key, Hewlett-Packard, Bellevue, Washington.

Figure 2. Large Floor Plan created with Sketch.

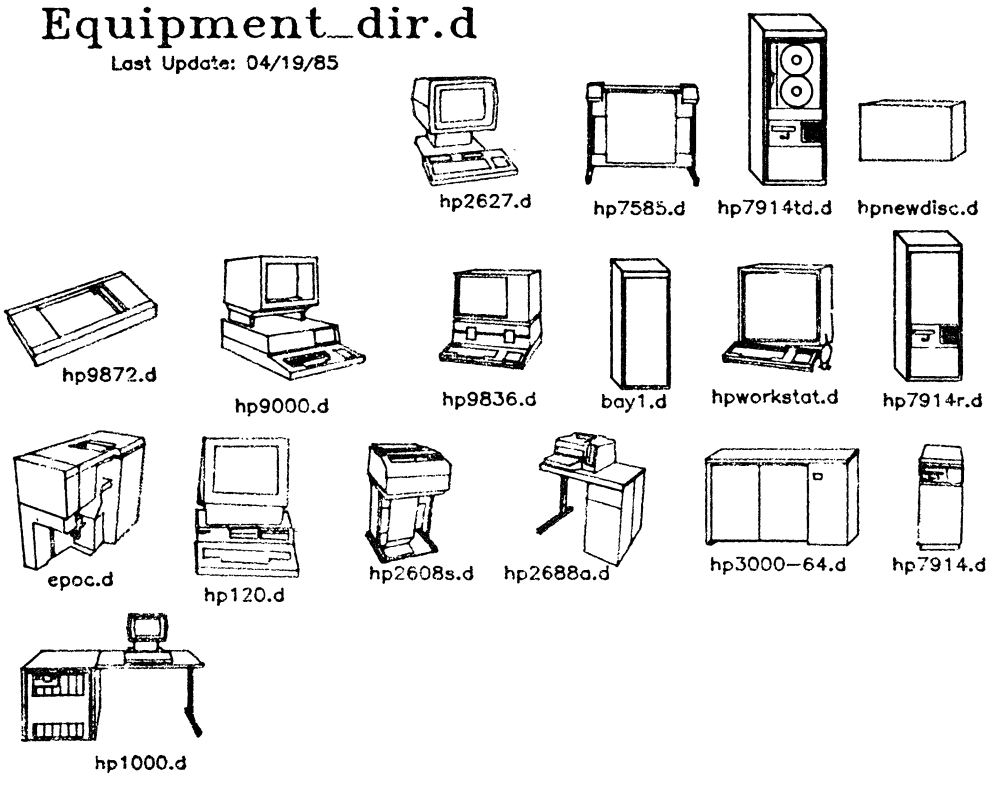


Figure 3. Library of Equipment Figures created with Sketch.

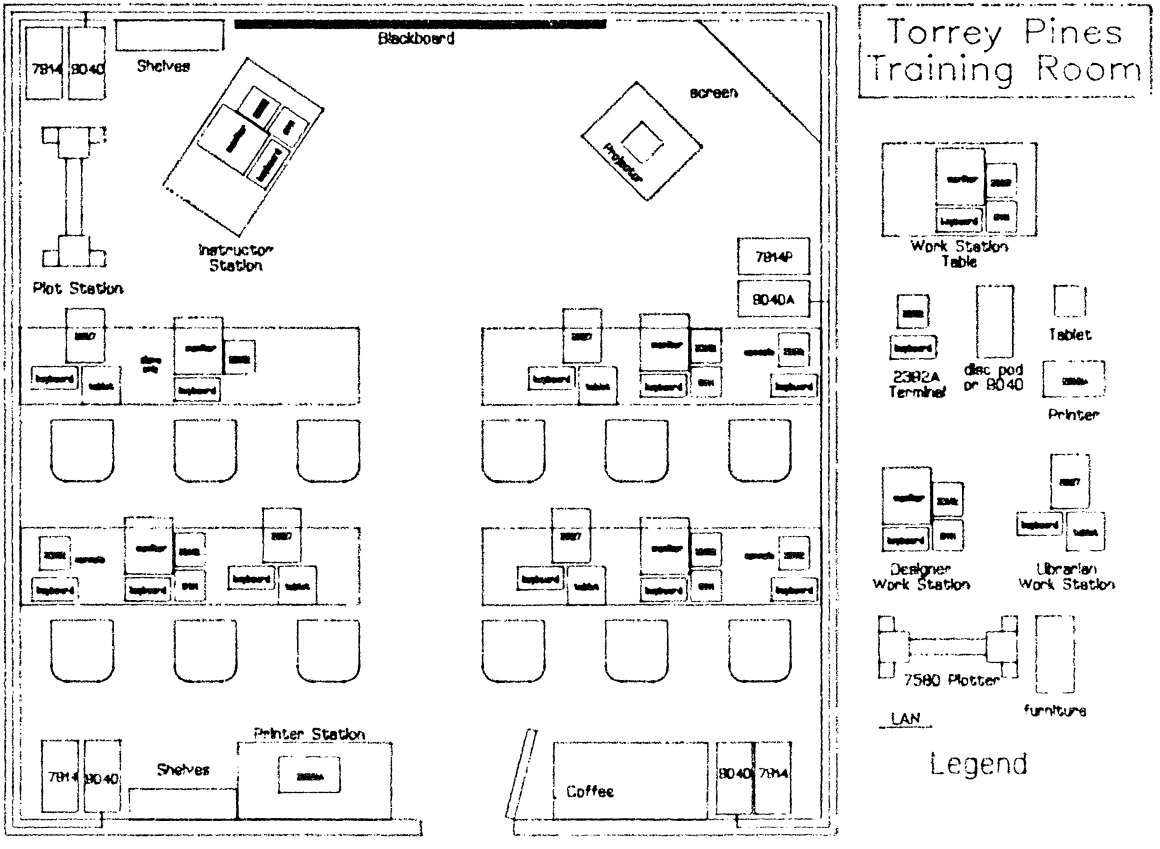


Figure 4. Local Area Floor Layout created with Sketch.

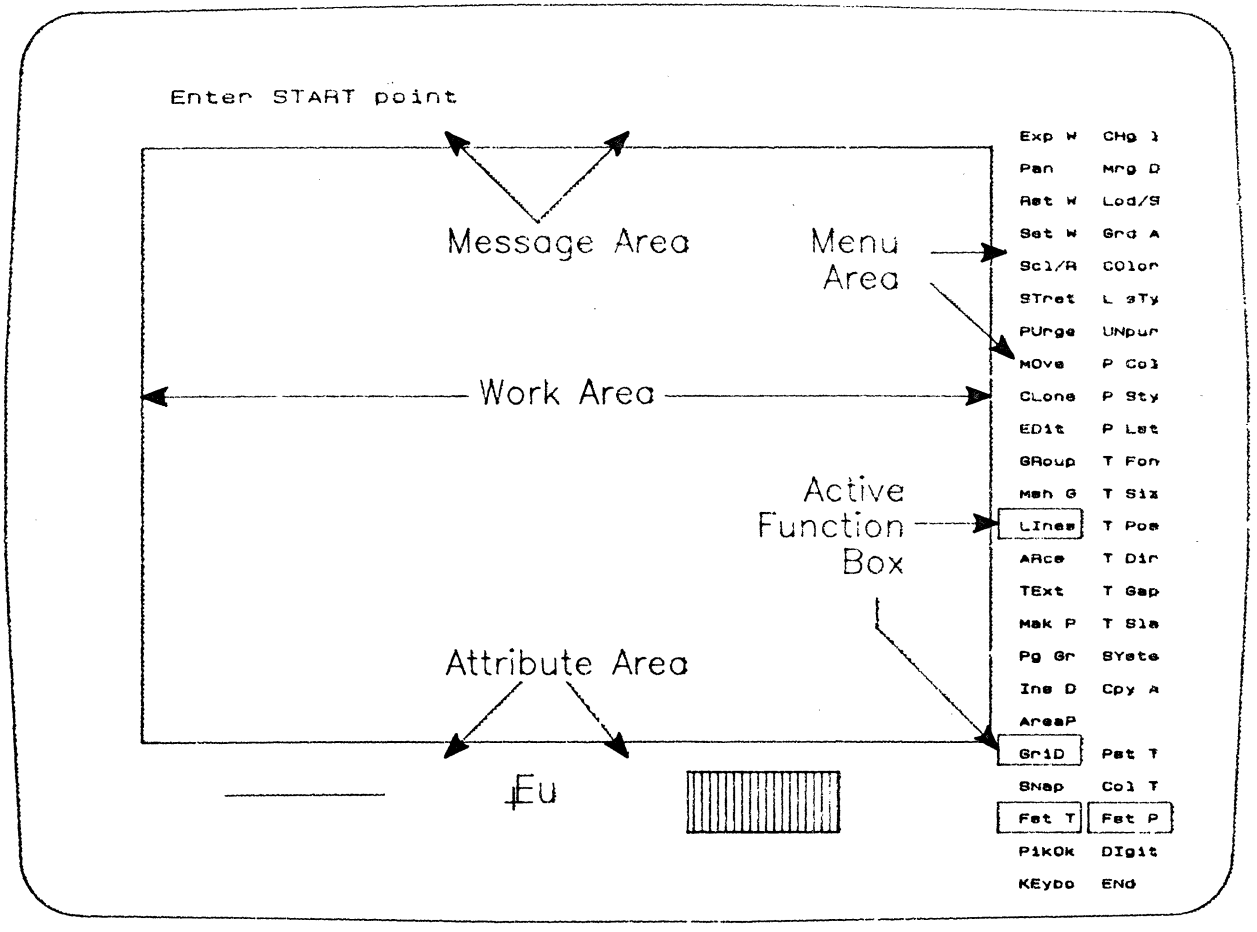
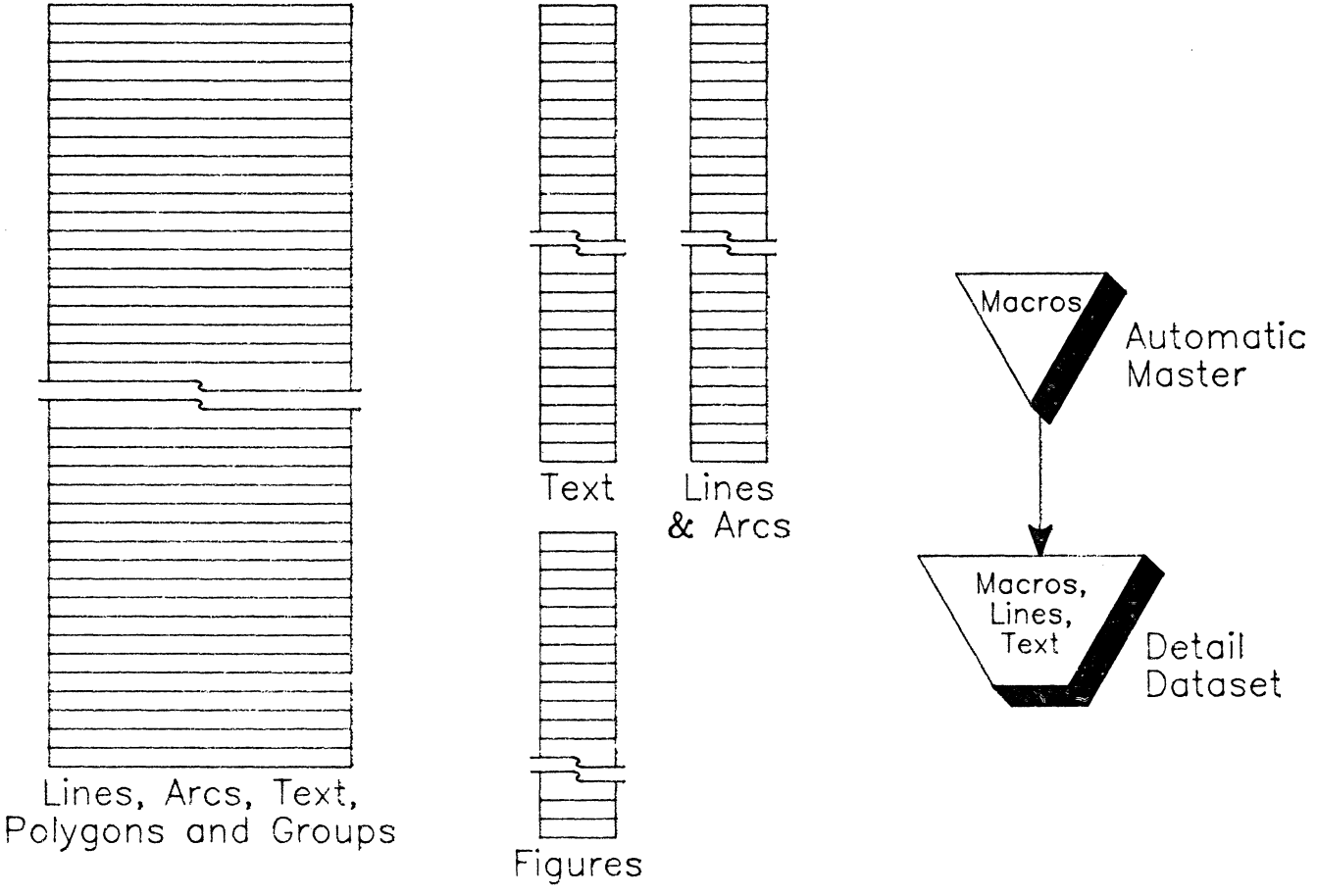


Figure 5. Layout of Display Screen for Sketch.



A. Sketch B. Bruno C. Penny

Figure 6. Sketch, Bruno and Penny Data Structures.

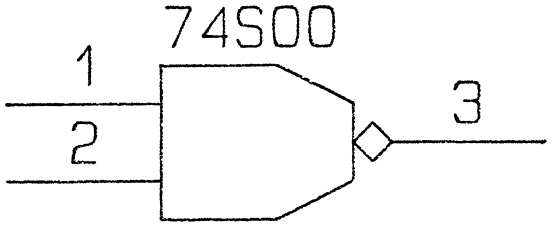


Figure 7. A Penny Drawn Nand Gate

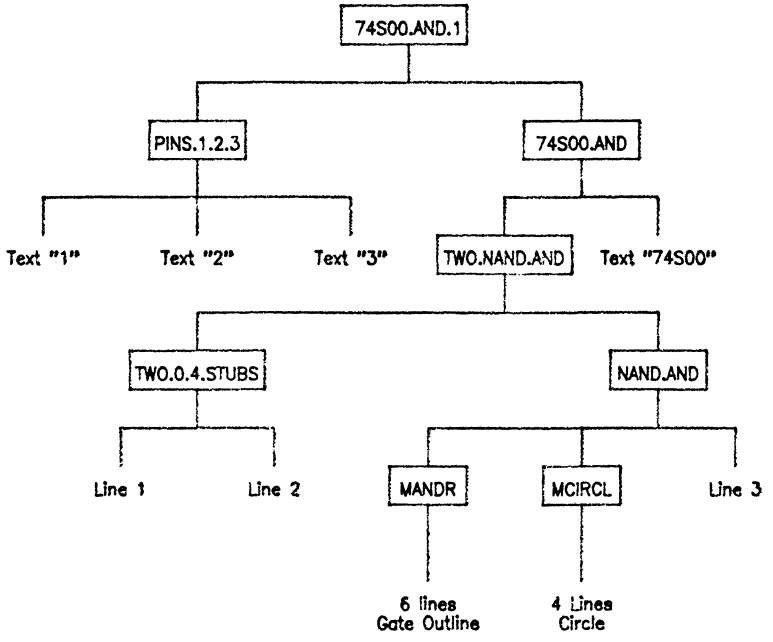


Figure 8. Macro Expansion of a Penney Nand Gate

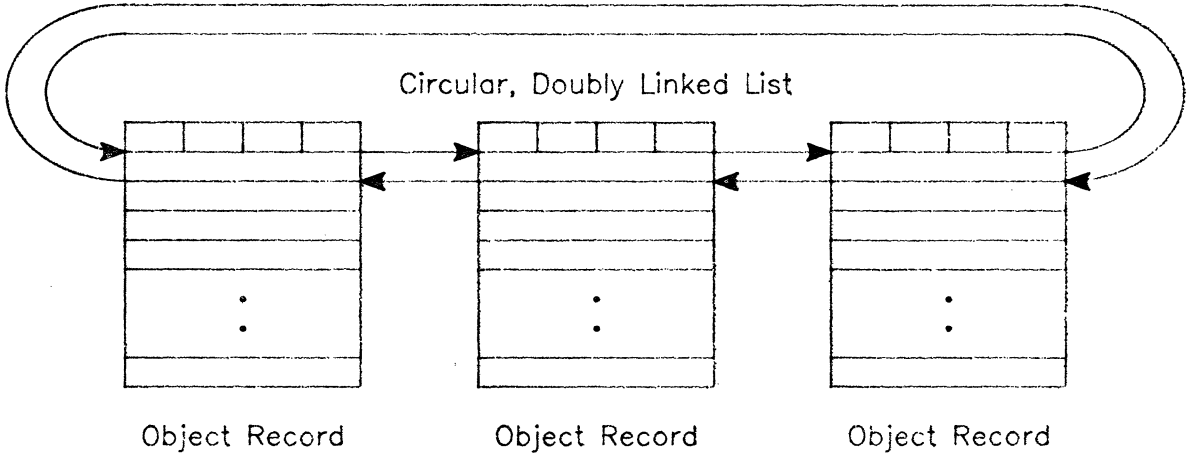


Figure 9. Structure of a Sketch Drawing.

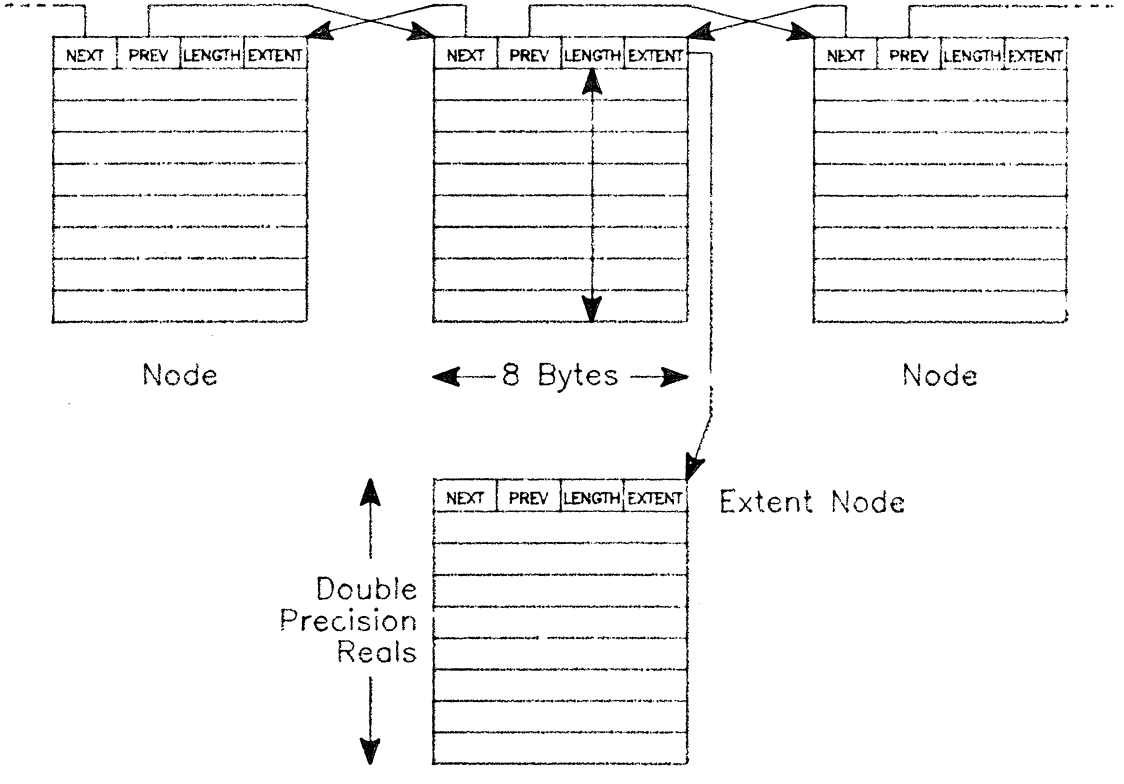


Figure 10. Sketch Node Structure

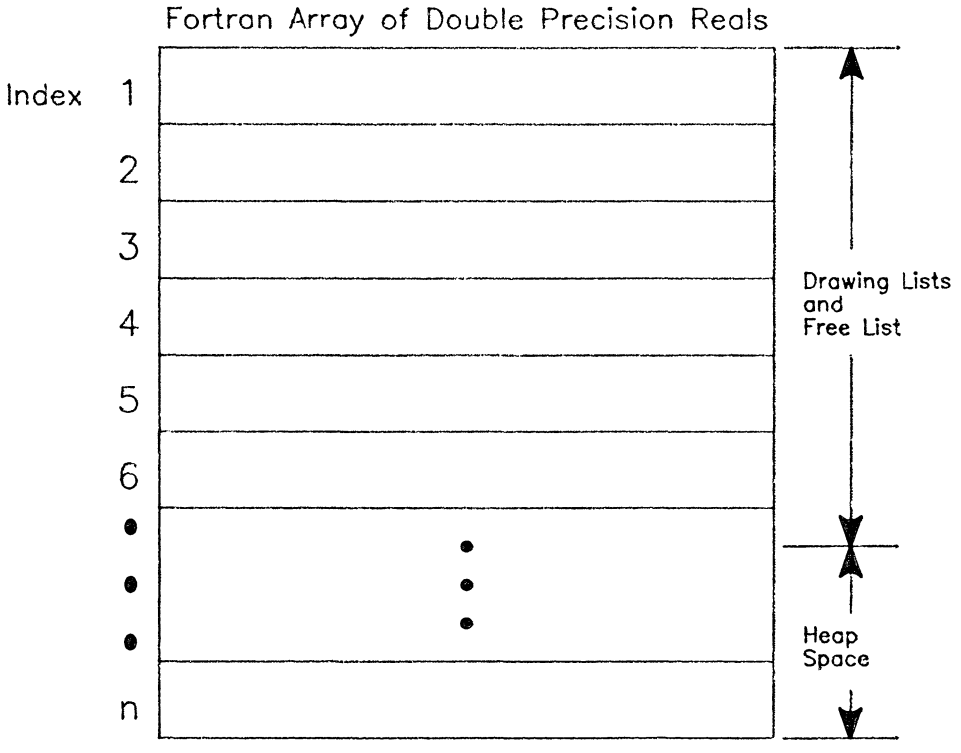


Figure 11. Allocation of Data Space.

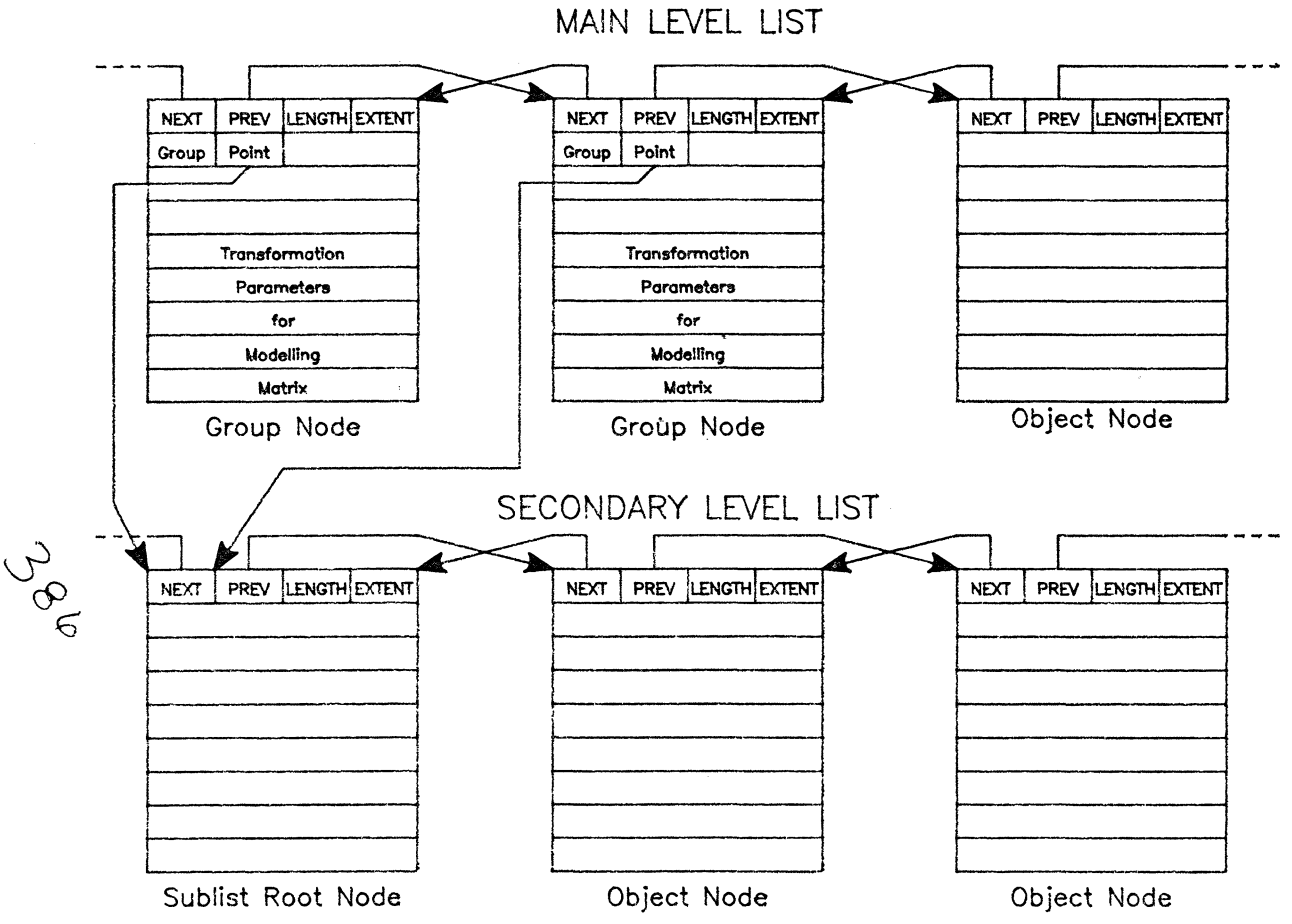


Figure 12. Two Group Nodes Referencing a Secondary List.

1031. EVALUATION OF HP REALTIME SYSTEMS PERFORMANCE

Martha Robrahn
Performance Technology Center
Information Technology Group
Hewlett-Packard
Cupertino, California, USA

PROJECT OBJECTIVES

The major objective of the SPEP project is to assist in the design and development of future systems and system components through the use of performance analysis technology. Specifically, this objective is being met by measuring system utilization and performance in actual customer environments. Tools to support this data collection effort and summarize the data are being developed. Data collected is used to model future system usage and to predict future system performance in order to facilitate design and implementation tradeoffs. The profiles of customers' usage of HP machines will also be used to develop standard benchmarks that emulate specific customer application types. These benchmarks can be used to characterize system performance of typical customer applications.

SPEP also collects patterns of customer usage that can be used to influence product decisions. What is the typical amount of memory and disc space on a system? What percentage of that space is used? What are typical buffer sizes for terminal transactions? For HP-IB transactions? These type of questions will be used to characterize customers' application workload mixes.

PROJECT BACKGROUND

In 1983, SPEP began collecting performance data on HP 3000 systems. To date, over 200 sites have been collected and summarized in a database. This data allows modeling of current and predicted performance of systems, as well as inquiry into typical customer configurations and workloads.

In 1985, a similar effort was initiated for the HP 1000 family of processors. Very little performance instrumentation or tools existed for the 1000 systems. ITG, DSD and selected field SEs worked together to implement the software necessary to fulfill the requirements of the SPEP program. An important side benefit of this effort will be the availability of system-wide performance data collection software to HP's technical field SEs. A subsequent section of this paper will cover the specific goals of the RTE-6 and RTE-A performance data collection efforts and describe what has been accomplished towards these goals.

Initial sites surveyed were selected by one of two criteria. DSD marketing selected those customers that they felt were representative of their customer base. Additionally, SEs were offered the opportunity to survey any customers they felt could benefit from the performance report generated from the data collected at their site.

GENERAL COMPUTER PERFORMANCE EVALUATION METHODOLOGY OVERVIEW

Computer systems performance evaluation and prediction can be divided into two components: performance measurement and performance modeling. Performance measurement is useful once a system has been implemented. Performance modeling can be used to predict performance of new systems as well as determining how a current system will perform when workloads or device configurations are altered. A workload is a grouping of similar processes consuming system resources.

Performance measurement techniques for real-time systems fall into three categories: hardware measurements, sampling techniques, and event driven measurements. Each type is useful in various situations.

Hardware measurements are typically taken using logic analyzers. They are used to determine the time it takes to do very specific things in HP's RTE operating systems: the time to enter a driver from an interrupt, the time to make a particular EXEC call, the time it takes to transfer n bytes of data via a particular driver/interface/device combination, etc. These types of measurements are performed under ideal conditions and typically represent best case numbers. They are found in the performance briefs [1,2] for RTE operating systems. They are useful when trying to determine whether a dedicated real-time application is a good fit for a particular processor and operating system.

Sampling techniques involve taking a snapshot of what is going on in the computer system at regular intervals. The code to do the sampling can be part of the operating system or simply a high priority privileged process. It could also be implemented with a logic analyzer. An example of a sampling program is HP's Profile Monitor.[3] This monitor observes only one program at a time, but the concept can be expanded to system wide monitoring.[4] Sampling can be useful when there are many long-lived processes on a system.

Event driven performance measurements require that the operating system be instrumented to log specific events as they happen. The result is a complete trace of the selected events during system operation or counts of the various events over the collection period. Typical events that might be logged are EXEC calls, program dispatches, I/O initiations and completions, swap traffic, VMA faults, etc. Obviously, this technique yields the most comprehensive performance information for a given system. It also causes the most overhead and could potentially yield misleading results if this overhead is too large. Event driven measurements should be most useful in characterizing real-time systems where many things happen too quickly for a sampling technique to detect.

Performance modeling techniques fall into two general categories: analytic modeling and simulation modeling. "A model is an abstraction of a system: an attempt to distill, from the mass of details that is the system itself, exactly those aspects that are essential to the system's behavior." [5] Models view the computer system as a series of servers (eg. CPU, disc) which provide resources to the different workloads made up of various programs. In order to model a computer system one must measure or estimate the inputs to the model. Inputs might include the amount of CPU time used by a process, the number and length of disc and other I/O operations, amount of time spent waiting for resources, etc. The outputs from the models include resource utilization, system throughput, and average response time.

Analytic models use mathematical algorithms to solve a set of equations to predict system performance. Simple system configurations lend themselves well to this kind of modeling technique. Analytic models are reasonably easy to define and evaluate. They execute quickly and therefore are useful for asking "what if" questions.

Simulation models attempt to mimic the computer system in detail. For complex systems, simulation may be the only accurate modeling technique. An example of such a system might be a data communications network. Simulations are complex to define and time consuming to run.

RTE DATA COLLECTION SOFTWARE IMPLEMENTATION

Keeping in mind the components of performance evaluation described above and the goals of SPEP, software was developed for performance monitoring of both RTE-6 and RTE-A systems. Brief descriptions of the instrumentation and reduction software follow.

It was decided to implement the software for RTE-6 outside of the operating system. The goal was to write the minimum amount of software that would allow analytic modeling of RTE-6 systems. Three events are captured during a performance run: program state changes, I/O initiations, and I/O completions. Additionally, the currently executing program is recorded with each event. From this information, a relatively complete description of program execution and RTE operation can be derived. The software that controls the data logging allocates a 1 KW block of System Available Memory (SAM), patches its code into the SAM block, and then patches RTE to call the logging software in SAM. Two additional programs are used to retrieve the event data from SAM and log it to disc or mag tape.

In parallel with the RTE-6 development, RTE-A was being modified to collect performance data. Some of the events that can be traced include those in RTE-6 plus VMA faults, EXEC calls, process activations and terminations, CDS faults, swapping, session logons and logoffs. Each event can be activated or deactivated individually. Event logging software was placed into RTE-A. Collection software was developed to log RTE and other events to disc or mag tape. (Use of this software requires a new system generation.) Additionally, sampling programs were developed to monitor SAM use, Id Segment use, Swap file use, class number use, memory use, resource number use and file system usage. This information will allow use of both analytic and simulation models. Additionally, it will allow access to many different types of information about how a particular system is being used. This should allow HP to better understand our customers' needs. It should also allow field SEs to be better able to deliver comprehensive performance consulting services to their customers.

A typical customer performance run involves sending a tape of software to the SE or customer. The files can be restored using standard utilities. The customer fills out a survey to more completely describe their application. A procedure file is used to load, install, and run the collection software. Customers were allowed to set the duration of the data collection run based on their specific applications. (For comparative purposes, all data in HP's summary data base of real-time customers will be normalized to one hour.) When the run is completed, the tape is returned to HP for reduction. A summary report is returned to the customer and the account SE. This report contains a summary of each program

executing during the data collection. It includes a list of the amount of time the program spent in each state as well as a summary of all I/O done by the program. Also contained in the report is a summary of disc usage.

SPECIFIC ISSUES RELATED TO REAL-TIME SYSTEM PERFORMANCE

There are many issues related to real-time system performance that do not seem to be of concern on commercial installations. They include the resolution of the time base generator (TBG), the applicability of modeling to real-time systems, and the definition of real-time workloads.

The resolution of the TBG on a 1000 is 10 milliseconds. Many events in RTE systems occur in less than this time. If three programs executed in one 10 ms interval, how should the CPU time be credited to those programs? Currently, the program executing when the clock ticks gets credited with the entire 10 ms. It is hoped that this will average out over time. However, with the time scheduling capabilities of RTE systems, this may be unrealistic.

Analytic modeling techniques work well for commercial systems. Most models tend toward system configurations that include only the CPU, discs and terminals. The assumption is that one process services one terminal. Will RTE programs that service multiple devices fit well into analytic models? How does one incorporate the large array of black boxes and instruments our customers use into these models? Will the applications using specific RTE features like class I/O, resource numbers, and Shareable EMA fit well in these models? Many of these types of questions need to be addressed.

Workload definition for our customers' real-time systems also seems to be more complex. Many commercial customers use "off the shelf" software packages to do much of their processing. As such, they are easily recognizable as a particular type of program: database, editing, accounting, manufacturing, etc. Most real-time customers write the majority of their own software. The grouping of programs into workloads will be difficult without an in-depth knowledge of individual applications.

Finally, there is the issue of what the criteria for acceptable performance is for our real-time customers. For a commercial customer, there are some very standard measures: terminal response time for interactive applications and throughput for batch applications. By definition, a real-time system must respond to external events "fast enough". The goal of performance optimization in real-time systems is to maximize throughput while insuring that real-time processing remains unimpeded. Our real-time customers may have some different measures of performance: interrupt response time, device I/O throughput, disc throughput, CPU speed, RTE overhead, etc. The survey sent to each customer participating in the SPEP program solicits their inputs on this subject.

CONCLUSIONS

Since data collection on RTE systems has just begun, there are no conclusions at this time. It is hoped that some preliminary results can be presented at the September conference.

ACKNOWLEDGMENTS

Special thanks to:

Bob French (HP Fullerton) and Dave Glover (HP San Ramon) for their work on the RTE-6 software. Also to Nick Copping (HP everywhere) for getting us into this.

Nigel Clunes (HP Sydney) who single handedly instrumented RTE-A and sold the concept to DSD. Also to Beth Clark (HP DSD) for believing that performance evaluation software would make an important contribution to RTE-A's feature set.

Tony Engberg and Paul Primmer (HP ITG) for their views that performance technology also belongs in the technical computer arena at HP.

Tony Lukes (HP ITG) for raising the state of the art of performance evaluation at HP to an ever higher level.

REFERENCES

[1] RTE-6 Performance Brief. HP part number 5953-2846. July 1982.

[2] RTE-A Performance Brief. HP part number 5953-8753. January 1984.

[3] RTE Profile Monitor. HP part number 92083A.
Manual part number 92082-90001.

[4] D. A. Thombs. "Dynamic RTE system monitors". TC Interface.
March, April 1985.

[5] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik.
"Quantatative System Performance". Englewood Cliffs, New
Jersey: Prentice-Hall, 1984.

[6] P. Heidelberger and S. Lavenberg. "Computer Performance
Evaluation Methodology". IEEE Transactions on Computers, Vol.
C-33, No. 12, December 1984.

1032. GOOD LABORATORY PRACTICES AND HOW THEY CAN BE SUPPORTED BY A LABORATORY INFORMATION MANAGEMENT SYSTEM.

Thomas C. Boyer, Kazmer Latven, Ph.D.
Scientific Instruments Div.
Hewlett-Packard Co.
1601 California Ave.
Palo Alto, CA 94304

Most of the applications that are written for the HP3000 address a concept called Generally Accepted Accounting Practices. Business data processing systems are audited by accounting firms that have standardized business controls and procedures. Laboratories are also audited but generally not by accounting firms. If a data processing system is to be built to support the laboratory, a concept called Good Laboratory Practices provides guidelines for the system designer.

Laboratories can be divided into two major classifications. The first category is a "project" laboratory. It is identified by the type of activities that are generally thought of as a project. Sub-categories would include an R+D organization, an Investigative organization, or a "Methods" development organization. The second category is a "process" laboratory. This type of laboratory handles samples as a process rather than a project. It is organized to handle many similar samples routinely. Sub-categories would include an in-house control laboratory in a chemical manufacturing plant, a similar type of process laboratory in a more regulated type of manufacturing process or a Contract laboratory which operates a laboratory as a routine business service. (See Figure 1.)

A typical project laboratory would be divided into groups organized around the chemist or engineers completing a focused project. These project teams have a number of laboratory services available to them that cross several disciplines. Although the time for the completion of a project is not fixed, this type of lab would expect to work on a project for a long time.

A typical process laboratory is divided into sections or cells that are organized to perform specific types of analysis. One area might be equipped to test for metals; another area, such as a mass spectrometer section, would be set-up to test for organic compounds. A sample receiving area is responsible for receiving the sample, assigning it a "method" and initiating the sample analysis process. (See Figure 2.)

Personnel functions that are to be found in a process laboratory include:

1) Operations manager

This is the business manager of the operation. He is responsible for the financial performance of the labs as well as the focus for customers and vendors. He bears the responsibility for passing a financial audit. He bills the customer for services. He is responsible for sample storage and disposal.

2) Lab manager

This is the technical head of the lab. He has some administrative responsibilities in the technical operation of the lab. He usually hires and fires the technical staff. He is responsible for the Standard Operating Procedures, the Methods, the qualification of the lab staff, and the equipment. He specifies sample storage and disposal requirements. He bears the responsibility for passing a GLP audit. He reports to the Operations Manager.

3) Chemist/ Engineer/ Technician

This is the person who actually does the productive work. Specifically, she has the education, experience and technical license to perform the tests and record the results.

4) Receiver/ Dispatcher/ Reporter

This is a key administrative function. This person receives the sample, assigns it a Method or process, passes the sample on to the proper station or inventory location, initiates the customer acknowledgment, initiates the billing, and distributes the sample report. He reports to the Operations Manager.

5) The Auditor/ Quality Inspector

This is either the person assigned by the regulatory agency to audit the lab for compliance or an in-house quality assurance person. The regulatory auditor does not check for financial performance but verifies that the lab's Standard Operating Procedures are adequate and are being followed. The in-house quality assurance person is responsible for inserting "spiked" and "blank" blind samples and for other operational quality control procedures. (See Figure 3.)

Other resources that are important to laboratories include the facility where the sample is processed and stored, the equipment that is used to perform the analysis, and the operating procedures the lab has implemented to control the process.

Many laboratories are audited by outside regulatory agencies. The Environmental Protection Agency, the Department of Defense, various health agencies, and the Food and Drug Administration all have specific requirements meant to insure that analyses that are performed generate valid results. While each of these agencies has similar requirements, there is no organization or society that provides inter-agency coordination of acceptable laboratory procedures. The FDA regulation entitled, "Nonclinical Laboratory Studies Good Laboratory Practice Regulations" (1),(2),(3),(4) abbreviated GLP, is becoming an industry standard that attempts to fill the vacuum left by a lack of coordination.

A major thrust of the GLP requirement is to have an audit trail of the resources used to perform an analysis and to be able to recreate the result based on archived record keeping. These audit trails might keep track of the calibration schedule of the instruments, the recent training of the laboratory personnel, or the version of the "method" that was used in the process. An auditor should

be able to trace back through the audit trail and verify that the conclusions generated from a test were valid. (5)

What follows is a short list of the tools common to the accounting or manufacturing systems designer that might be used in a Laboratory Information Management System that supports GLP.

Besides the normal security associated with the MPE operating system, other security systems such as a positive identification system might ensure that the operator of the instrumentation would find a match in a list of qualified operators.

Automatic machine-to-machine communication not only offers speed but reduced errors. Besides the obvious advantage that the computer system brings in automating the laboratory, there is less chance for errors when the results of an analysis are directly read from a file.

Machine logging of transactions is very familiar to the HP3000 application designer. Besides machine logging, database logging provides a strong archiving tool. Archive retrieval is supported also.

Existing applications also offer some aids. Bar codes and check digits ensure the proper sample disposition. Reasonableness checks in screen forms packages reduce errors.

Text editors that keep track of versions of software code should be easily applied to handle versions of methods and processes.

1) "Nonclinical Laboratory Studies Good Laboratory Practice Regulations," FEDERAL REGISTER, Vol.43, December 22,1978, pp. 59986-60025.

2) "Good Laboratory Practice for Nonclinical Laboratory Studies; Amendment of Good Laboratory Practice Regulations," FEDERAL REGISTER, Vol. 45, April 11, 1980,p 24865.

3) "Review of Agency Rules," FEDERAL REGISTER, Vol.46, July 14, 1981, pp. 36333-36335.

4) P.D. Lepore, M.M. Kemp, "FDA's GLP regulations: proposals for change," PHARMACEUTICAL TECHNOLOGY, January 1984, pp. 30-36.

5) King, Paul G., "Laboratory Computerization and Good Laboratory Practice Standards (GLPS)," ANALYTICAL INSTRUMENTS & COMPUTERS, May/June 1984, pp. 14-15.

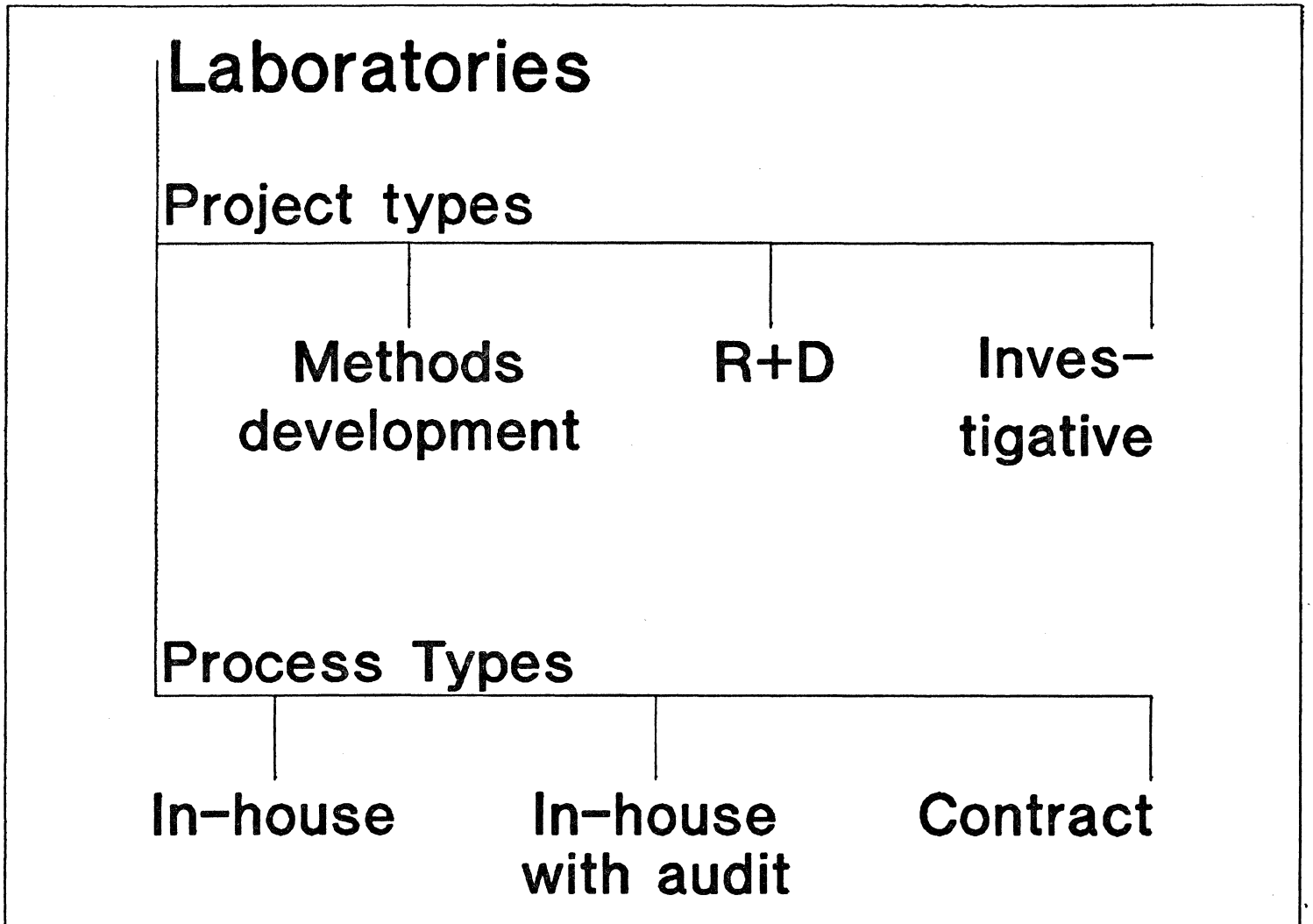

Laboratories
Project types
Methods development
R+D
Inves-tigative
Process Types
In-house
In-house with audit
Contract

Figure 1

Typical Lab Layout

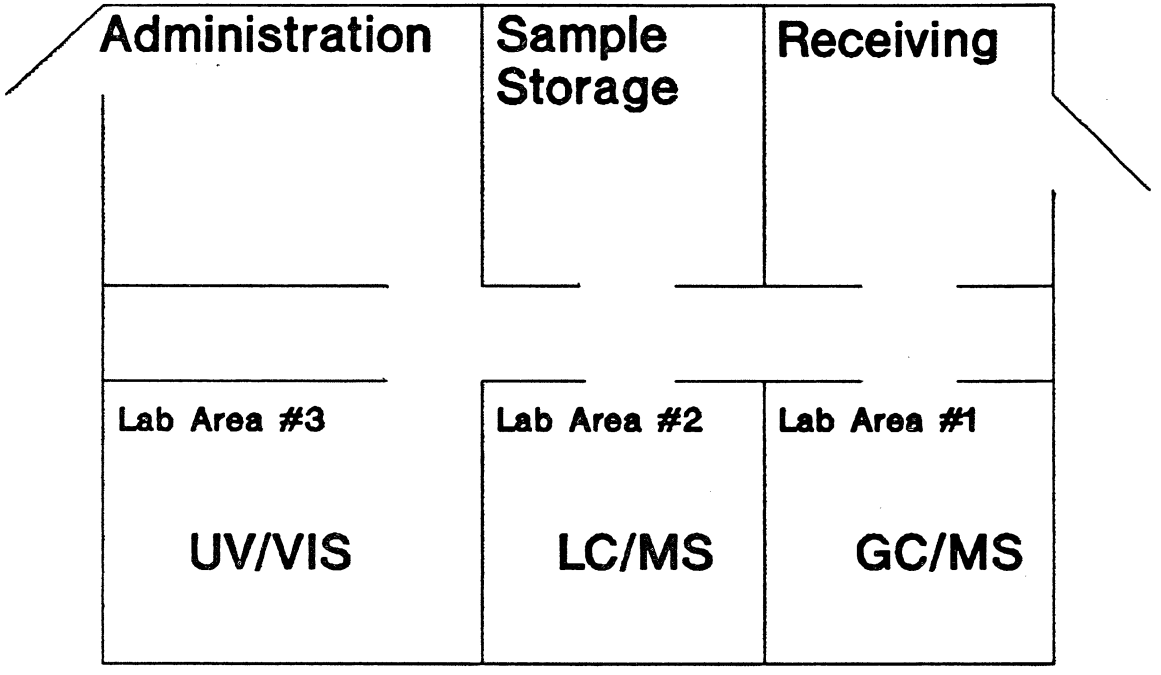


Figure 2

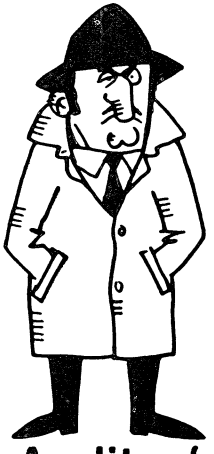
Typical Lab Functions

Operations Manager

Chemist/ Engineer

Receiving/ Dispatch

Lab Manager



Auditor/ QC

Figure 3

1033. TROUBLESHOOTING STRATEGIES FOR HP 1000 COMPUTER USERS

Lizette I. Mill
Hewlett Packard Co.
11000 Wolfe Rd.
Cupertino CA 95014-9974

Troubleshooting for Supportability

Programming for supportability is a topic that has been widely discussed in the computer science field. Students learn structured programming methods in university courses, and programmers are cautioned to document extensively and to use "goto-less" programming techniques.

On the other hand, troubleshooting (problem resolution) techniques have not been given the same attention. This is unfortunate, for it becomes obvious after even a cursory study of problem resolution and design that these activities, normally regarded as distinct from one another, are simply two guises of the same process.

The HP 1000 factory support group has a running joke that any HP 1000 troubleshooting problem can be resolved with one of the following answers:

1. Halt and reboot,
2. That's not supported,
3. That's fixed in the next revision,
4. Read the manual, or
5. That's not a bug, that's a feature.

Although we treat it humorously, like any myth, this joke has a certain ring of truth.

The HP 1000 computer is, by design, an open system. Manipulation of the system is a programming feature. This means that users can tie their systems into programmatic Gordian knots by incautious use of EXEC calls, rash exercise of privileged code and other unparalleled feats of daring. The solution of cutting the knot by rebooting does not actually solve the problem. The programmer needs a better answer than that provided by Alexander the Great's straightforward method of problem resolution. The programmed Gordian knot usually holds the whole system together.

I cannot cover here what I have learned about problem resolution techniques in two years of troubleshooting in the HP 1000 factory support group. That would take two years. Moreover, it would require the reader to attend (then teach) HP 1000 internals courses, answer obscure questions over the phone for weeks on end and resolve knotty system problems using logic analyzers and assembly language programming. But, I can pass on some tools and techniques which I have found especially useful during those two years.

Methods and Mechanisms

The most important troubleshooting tool is the mind of the troubleshooter. Without a systematic, comprehensive approach to the troubleshooting process and the ability to use problem-solving techniques, the HP 1000 computer system's information sources will simply drown the troubleshooter in a sea of details.

James L. Adams in his fascinating book "Conceptual Blockbusting" describes the problem-solving technique we are all prone to use.

> *(T)he natural response to a problem seems to be to try to get rid of it by finding an answer-- often taking the first answer that occurs and pursuing it because of one's reluctance to spend the time and mental effort needed to conjure up a richer storehouse of alternatives from which to choose. This hit-and-run approach to problem-solving begets all sorts of oddities-- and often a chain of solution-causing-problem-requiring-solution, ad infinitum. In engineering one finds the "Rube Goldberg" solution, in which the problem is solved by an inelegant and complicated collection of partial solutions.*

Adams goes on to describe blocks that obstruct the design and problem-solving processes, and conscious and unconscious methods of "block busting". He suggests using different problem-solving languages: verbal, visual, mathematical and diagrammatic. He recommends different problem-solving strategies: elimination, substitution, comparison, separation, dissection, list making, and so on. These are all valuable problem-solving techniques. But the most important element of the troubleshooting process is a consistent problem analysis method.

A number of authors have written on the subject of problem analysis. Some, such as Kepner-Tregoe, Inc., teach courses on it. But, as William J. Reilly pointed out in 1947, one of the most powerful problem analysis techniques ever devised has been around for quite some time. That technique is the scientific method, which Reilly described as follows:

1. *Some kind of observation suggests an unanswered question or a problem.*
2. *The problem is defined and possible solutions considered.*
3. *Controlled experiments are set up and observations made.*
4. *A conclusion is drawn based on the results of experimental and other observations.*

Reilly noted that to reach the second step of the scientific method, you must expand the original observation to obtain a precise understanding of the problem. This expansion is contracted into a problem definition. The problem is expanded again as more observations are made in the third step to determine the validity of each possible solution. Finally, the process contracts to a chosen solution.

Figure 1 illustrates this process.

It's fairly obvious that we go through this process every day. We observe that the light has turned yellow. We define the problem as the fact that our car must not be in the intersection when the light turns red. We consider possible solutions: we could speed up in hopes of getting through the intersection before it's too late, or we could slow down and stop before entering the intersection. We've done experiments in the past that provide data on how the speed of the car and the length of the yellow light affect the outcome of these possible solutions. Based on these past experiences and other observations, such as how late we will be for work and the speed of the car behind us, we draw a conclusion and choose a solution.

It's also true that we fail to apply this technique every day. We observe ants marching resolutely into the bathroom from under the floor board. We spray the floor board. The next hour finds the ants marching in from behind the sink. We spray behind the sink. The next time the ants are coming out from under the bath tub. Finally, in desperation, we call the exterminator. He's been troubleshooting insect problems for years. He's a firm believer in the axiom, "Garbage in, garbage out." He knows the ants are coming from somewhere. So, he goes out the back door and sprays where the ants are marching in under the outside spigot. No more ants. If we had applied the scientific method to this insect problem, we could have been spared the $50 exterminator fee.

Let's see how this process might be applied to a simple HP 1000 troubleshooting situation.

Tom Programmer occasionally gets FMP -11 errors from his program PlotCurve. He observes the occurrence for a time and deduces that PlotCurve only gets FMP -11 errors if it's run at the same time as another of his programs, PlotHisto. Tom's a bit intimidated by all the packages he's been using to write these programs. He's not sure they all work together the way they should. Instead of using a systematic approach, Tom examines the two programs hurriedly and notices that they both use system common.

"Ah, ha!," Tom speculates prematurely, "it must be that PlotHisto is somehow corrupting PlotCurve's blank common. I'll bet that PlotCurve is corrupting PlotHisto's blank common, too, but the symptoms aren't as obvious." Tom rushes to the phone and calls Sharon Troubleshooter over at the HP 1000 factory support group.

"Sharon, I've found a problem where two programs are writing into each other's blank common," he explains. "The blank common data in one program is getting corrupted when I run another program that uses blank common."

Now, Sharon's been troubleshooting HP 1000 problems for some time. She is immediately skeptical. The two programs are running in completely different areas of memory. Moreover, that area of memory will probably be different each time the programs are dispatched. She's learned the value of the systematic approach, so she starts leading Tom through the process.

1. Some kind of observation suggests an unanswered question or a problem.

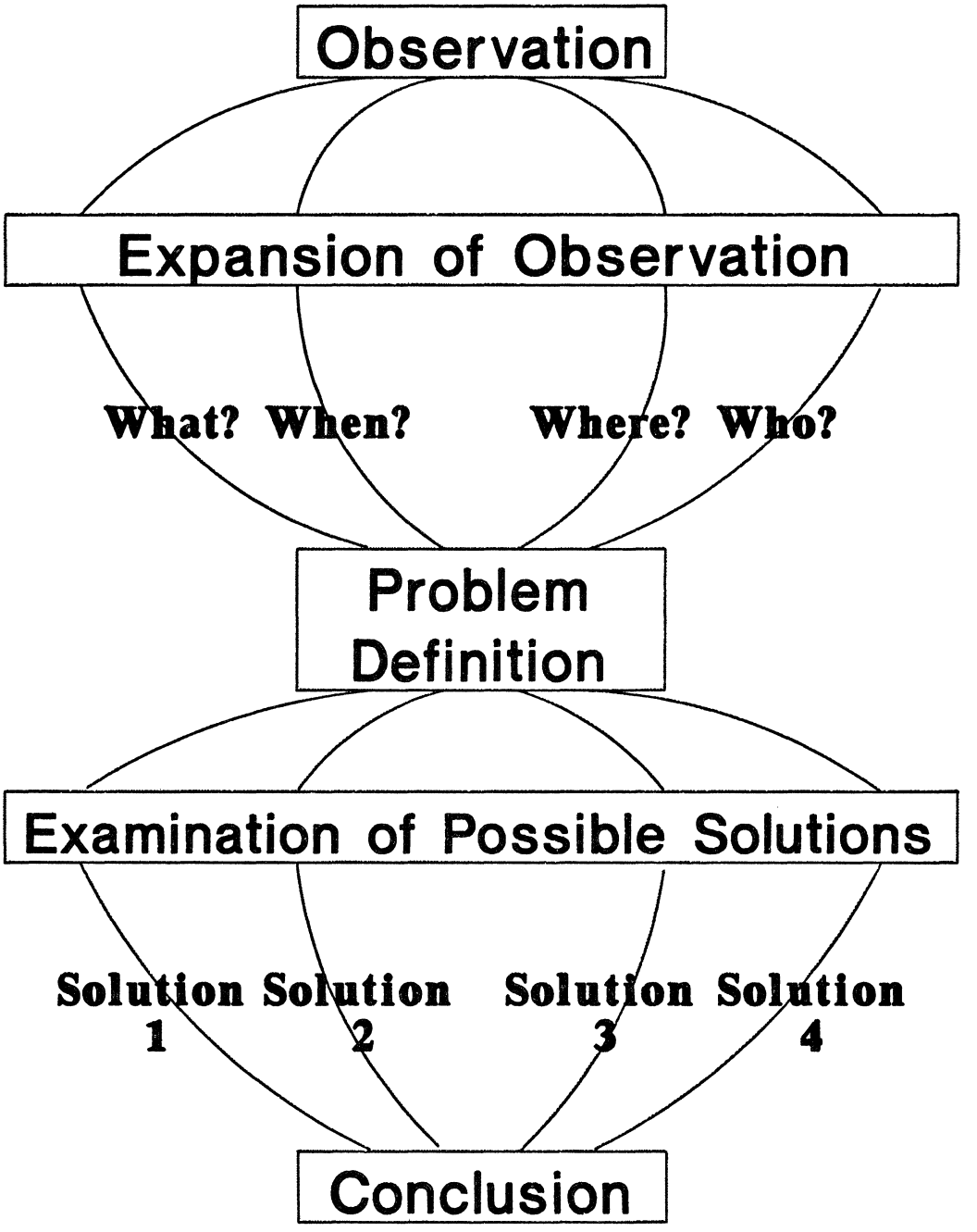


Figure 1: The Scientific Method

Sharon: "What error are you seeing, Tom?"

Tom: "I'm occasionally getting FMP -11 errors."

2. The problem is defined and possible solutions are considered.

Sharon wants to define the problem more precisely, so she starts considering the famous W's: Where, What, When, Who.

Sharon: "When do these FMP -11 errors occur?"

Tom: "My program PlotCurve gets them when I run another program, PlotHisto."

Sharon: "Are the FMP -11 errors occurring only with PlotCurve, or are other programs exhibiting the same failure symptoms?" (Who gets this error?)

Tom: "So far I've only seen it with PlotCurve, but I've only got these two programs running at the same time. Later, I'll have more programs running simultaneously, and they'll use blank common too, so they'll have the same problem, I'm sure."

Sharon: "It's possible. What does PlotCurve do, anyway?"

Tom: "Well, I'm using Image/1000 and the DGL Graphics package to plot graphs of collected data."

Sharon: (looks in the Quick Reference Guide) "An FMP -11 means DCB not open. Does your program explicitly use FMP calls, or are they called by Graphics or another routine?" (Who calls FMP?)

Tom: "I use FMP explicitly to read data from a temporary file."

Sharon: "That's interesting. What's the format of your FMPOpen call?"

Tom: "The call is 'type = FMPOpen(dcb,error, plotfile,'ros',3)'."

Sharon: "Your options are 'ros'? You're using a shared file?"

Tom: "Yes, I am."

Sharon: (the light's dawning) "Is PlotHisto also accessing that shared file?"

Tom: "Yes."

Sharon: "Does one of the programs close that file by any chance?"

Tom: "Well, they both close the file on termination."

Sharon: "Hmmmm. Does this problem occur only after one of the programs terminates?"

Tom: "Now that you mention it, it might work that way. I'm not really sure."

Sharon: "Well, it sounds to me like a problem closing a shared file. Let me check if there's any known bugs on that, and call you back."

3. Controlled experiments are set up and observations are made.

Sharon can't turn up any problems with closing shared files in her bug database and R&D hasn't heard of anything like it, except possibly an old bug fixed in the last revision. When she calls Tom back to find out if he has old software, he has solved the problem. He had neglected to FMPClose the shared file in PlotHisto. If PlotHisto closed the file, the problem went away. By using better questions, he was able to come up with a more likely cause than the blank common theory. Research into the more likely theory produced the solution to the problem.

4. A conclusion is drawn based on the results of experimental and other observations.

Tom explicitly closes all his FMP files now, especially the shared ones.

Steps 2 and 3 are the two most important parts of the troubleshooting process. Problem definition and experimentation (isolation of cause) are essential to accurate and comprehensive troubleshooting. Figure 2 breaks down the problem definition process (step 2 of our scientific process) for HP 1000 problems. The appendix includes a form that can be used during this process.

The fourth problem definition step in Figure 2 is particularly important for the very effective troubleshooting technique of isolation. Reproducing the problem symptoms with the smallest possible set of problem inducers often pinpoints the cause.

For example, reports from the field alerted the HP 1000 factory support group that I/O to certain IEEE-488 devices would fail if the FORTRAN subroutine LGBUF (use a large buffer) was called with a buffer larger than 128 words. After the

1. Define the environment.

   For an application program, define the subsystems being used-- FMP, Graphics, Fortran, Pascal, a database package, system library routines.

   For a system problem, define the CPU (A900, A700, A600+, E-series, F-series), the operating system (RTE-A, RTE-A/VC+, RTE-6/VM, RTE-4B, a memory-based system), peripherals (printers, terminals, plotters, black boxes), interface cards (HPIB, multiplexor, ASIC, DS).

   Check your revisions! Many problems are caused by the accidental mixing of revisions (whether your own, HP's, or another vendor's). The problem symptoms are usually obscure and bizarre.

2. Define the symptoms.

   This calls for the W's. What are the symptoms? What are the meanings of any error messages that are generated? When do the symptoms occur? What is happening at the time that the symptoms occur? Where (in the program, in the operating system) does the problem occur?

3. Define the purpose of the application, if appropriate.

   What is the intended task? Is the application attempting to perform this task "legally"?

4. Determine the history.

   When did the symptoms start occurring? Did anything change, and when did this change occur? Did the environment change (new hardware or subsystems), and when? What has already been attempted while trying to solve the problem? Has it ever worked?

Figure 2: The Problem Definition Process

problem was isolated to any I/O transmission greater than 128 words between the HPIB interface card and a particular class of devices, the theory that FORTRAN was at fault could be discounted and the actual problem cause deduced. Above 256 bytes the HPIB card terminates data transfers using the EOI bit. Some devices can only terminate using CR/LF. Data can only be transferred in 256-byte blocks across that interface card to those types of devices. This fact simply needed to be well-documented.

Troubleshooting Tips

There's a few quick sanity checks that should be performed during the HP 1000 troubleshooting process.

1. Read the appropriate documentation if you're not absolutely certain that you've done things correctly. See the appendix, section A.2, for a list of useful documentation.

2. Check the Software Status Bulletin for known problems. Scrutinize the problem descriptions. The unwary may waste time chasing false leads, while the cautious troubleshooter uses the SSB to avoid unnecessary troubleshooting on a known problem. See section A.2 of the appendix for information on the SSB.

3. Check your generation. Look for correct revisions and proper configurations. This was the solution in the case of the Puzzling Shared Program Problem. When Cari System-Manager tried to run her new shared version of CI, the system informed her, "Can only run unshared CI." Cari checked the index of her RTE-A System Generation and Installation Manual and found a shared programs specification command. Her current generation specified zero for the number of shared programs in the system. She regenned with one for the number of shared programs and solved the case. Elementary.

Tools and Techniques

There are a variety of information sources and troubleshooting tools available for HP 1000 problems. I'll briefly review those that I've found particularly valuable, along with some pointers on their usage.

Keep in mind that RTE-A and RTE-6/VM are table- and list-driven operating systems. Status utilities obtain their information from these tables and lists. System troubleshooters may want to examine those tables and lists directly, while application programmers obtain ample information for their troubleshooting needs from the status utilities alone. Use of the tools discussed below should be tailored to the problem at hand.

1. Symbolic Debug/1000

   This HP product is a powerful application program troubleshooting tool. A data sheet describing its capabilities can be found in section A.3 of the appendix.

2. Load map

A program's load map is obtained using the LL command of LINK or by specifying a list LU or file namr as a runstring parameter of LOADR. The load map is most valuable when used with a listing from FORTRAN, Pascal or Macro that includes the relocatable address of each instruction.

In FORTRAN, this type of listing can be obtained using the Q compiler option or the M compiler option. (See the FORTRAN 77 Reference Manual, HP Manual Part Number 92836-90001, Chapter 7.) In Pascal, the equivalent compiler options are the CODE_OFFSETS ON compiler option and the KEEPASMB compiler option. (See the Pascal/1000 Reference Manual, HP Manual Part Number 92833-90005, Appendix C.) In Macro, the relocatable address is provided in the second column of the listing. (See the Macro/1000 Reference Manual, HP Manual Part Number 92059-90001, Chapter 1.)

When using the FORTRAN Q and the Pascal CODE_OFFSETS ON options, keep in mind that each high-level language statement is equivalent to several relocatable instructions, so that the relocatable address for one statement might be 117 octal, while the next statement is 130 octal. This means there are 11 (octal) relocatable instructions needed to perform the action of the first statement. In Macro, there is generally a one-to-one correspondence between line number and relocatable address, though there are exceptions for comment lines and so on.

How is this information useful? Let's look at an example. Figures 3 and 4 show the program TEST's listing and load map.

When an error message (such as a system error message) provides a P-counter value, this value can be calculated by adding the relocatable address from the compiler listing to the module start address from the load map. Using our example, if a system error message listed "2124" (octal) as the offending P-counter value, we can determine that the error occurred in statement 5 of our Pascal program.

```
1  0  : $KEEPASMB
2  0  : Program Test(input,output);
3  0  :
4  0  1 BEGIN
5  0  1    writeln ('This is a test.');
6  0  1    writeln ('This is another test.')
7  0  1    END.

                          C O D E   O F F S E T S

        STMT OFFSET    STMT OFFSET    STMT OFFSET    STMT OFFSET

          5  00117       6  00130       7  00141

                0 Errors detected.
                0 Warnings issued.
               12 Source lines read.
              597 Words of program generated.
```

Figure 3: TEST listing

```
 4:20 pm  Jun  6, 1985
Purging old file: TEST.RUN:::6:89
Load map:
PAS.GLOBALINFO      2000     0. 850606.1536
TEST     2000   597. 850606.1536
           .
           .
           .

FMPBUFFER          25315    64. named common
FMPREGS 25415     2. named common

Main       2000 - 25416    9999. words

Program TEST.RUN:::6:89 ready; 11 pages
 4:20 pm  Jun  6, 1985
Runnable only on an RTE-A system
```

Figure 4: TEST Load Map

3. System Generation Listing

In addition to configuration information, the gen listing provides address information for system modules and drivers. This address information can be used as described above for a program load map. The only difference is that the gen listing would be used with operating system and driver code listings, rather than application program listings. This is useful if you are troubleshooting your own driver (or the operating system, as we do in the factory support group).

If you are on HP support services, make a practice of having your gen listing handy when calling HP. HP support engineers will probably ask you to check your gen at some point in the troubleshooting process.

4. Status Utilities

IO and LUPRN are the I/O status utilities on RTE-A and RTE-6/VM, respectively. They provide information about the devices on your system, including the LU number, the type of device and the device's availability. Chapter 6 of the RTE-A User's Manual (HP Manual Part Number 92077-90002) describes the IO command, and Chapter 2 of the RTE-6/VM Utility Programs Reference Manual (HP Manual Part Number 92084-90007) describes LUPRN.

WH and WHZAT are the program status utilities on RTE-A and RTE-6/VM, respectively. Depending on the options specified, they provide information about programs and memory usage on your system, including program name, partition usage, program size and program status. There are more options for WH on RTE-A than for WHZAT on RTE-6/VM. WH is described in Chapter 6 of the RTE-A User's Manual, while WHZAT is described in Chapter 2 of the RTE-6/VM Utility Programs Reference Manual.

There are two additional useful status utilities on RTE-A: METER and SAM. METER is most useful for system profiling. It has one advantage over WH as a status utility. METER continually displays information that is updated about every 5 seconds. This is useful for characterizing system activity during an error occurrence. SAM lists information about System Available Memory utilization, including free SAM information. SAM can be scheduled programmatically, with the SAM information returned in the EXEC or FMPRunProgram parameters. These utilities are described in Chapter 9 of the RTE-A Utilities Manual (HP Manual Part Number 92077-90004).

RTE-A with VC+ also provides an error logging capability. Through the spooling system, the user can direct the system to echo the error messages that appear on the system console to a file or a hardcopy device. Again, this can be useful for characterizing system activity during an error occurrence. Error logging is described in Chapter 2 of the RTE-A User's Manual.

5. M/E/F-Series Computer Front Panel

The physical front panel of the M/E/F-Series CPU displays register contents and can be used during operating system and driver troubleshooting to examine and modify system values.

Some especially useful activities are:

a. examining the P, M and T registers in standard register mode to obtain information on where a problem is occurring,

b. examining the m, t and f registers (dynamic mapping registers) in special register mode to determine what map the system is executing in, the contents of the map registers and other dynamic mapping system information, or

c. patching values in the system, such as changing an instruction just before a potential problem area into a halt so that relevant system or driver values can be checked, or changing a table value to observe the effect on the problem symptoms.

How is this done? By pushing switches and observing lighted indicators. The HP 1000 M/E/F-Series Computers Technical Reference Handbook (HP Manual Part Number 5955-0282) Chapter 2 describes how to use the switches, while Chapters 3 and 4 explain the meaning of the lighted indicators.

6. The A-Series Virtual Control Panel

The A-Series' Virtual Control Panel allows you to do everything that can be done with the M/E/F physical front panel. The difference is that VCP is both simpler to use and more powerful. Moreover, you read octal numbers, rather than lighted indicators.

VCP can be used to examine all the important registers. The VCP commands are described in Chapter 2 of the appropriate A-Series Computer Reference Manual (see the appendix, section A.2, for information on these manuals).

VCP has a particularly valuable command if you are working with HP on a troubleshooting problem. That command is the %W command. This command can be used to dump the contents of memory to magnetic tape or Linus cartridge for analysis by HP support engineers. Let's discuss how this command would be used through an example.

Our example system has 512K bytes of memory. The device to which VCP will dump memory is a magnetic tape drive at HPIB address 4, select code 27. The memory dump would be performed as follows:

a. Inform VCP that memory size is 512K bytes. This is accomplished by setting memory location 0 to the number of 64K byte memory blocks and memory location 1 to the number of words in the remaining memory block that is less than 64K bytes.

   Use the RW command to insure that the dynamic mapping system is pointing to the system map: that is, set the value provided by the RW command to 0. Then, use the M command to set the current location to 0. Now use the T command to change that location's contents to 10 octal. This informs the system that there are 8 (decimal) 64K byte blocks in memory (512K bytes). Since 512 happens to be divisible by 64, location 1 should be set to 0. Again, use the M and T commands to do this. This is what the process might look like (user inputs are underlined):

```
VCP> RW 000002 0
     RW 000000
VCP> M 001531 0
     M 000000
VCP> T
     M 000000 T 000003 10
     M 000000 T 000010
VCP> M 000000 1
     M 000001
VCP> T
     M 000001 T 000060 0
     M 000001 T 000000
```

b. Direct VCP to dump to the magnetic tape drive. (You've put a tape on the drive, of course.) The %W command format is similar to the format for the %B (boot) command. In our example, the command would be:

```
VCP> %WMT4027
```

7. The 1630 Logic Analyzer

This troubleshooting tool is far too powerful for most users' problem resolution needs. However, during system-level troubleshooting such as driver debugging, the logic analyzer is extremely effective for pinpointing problem code, especially when used with the M/E/F-Series' front panel or the A-Series' VCP. The 1630 analyzer monitors hardware signals and can be configured to trace and store instruction fetches.

HP currently offers 1630 analysis interface cards for the M/E/F-Series CPU, for the A600/A600+ CPU and for the A700 CPU. An analysis interface card for the A900 should be available by late summer of 1985.

8. System Analysis Software

System analysis software tools find and display system tables, lists and entry points. They may also have the capability to modify memory or disc, search memory or disc for a specified value, perform comparisons on the contents of memory locations, manipulate maps, dump memory and produce hardcopy listings of system information.

Some system analysis tools are available through the Interex contributed library. You can also write an analysis utility tailored to your needs. An interesting approach is to write such a utility as a driver. This puts the code in the system map where system tables can be easily accessed. The code can be entered via an EXEC control request through the dummy LU associated with the driver.

It's Only A Joke

Let's re-examine our HP 1000 troubleshooting joke. Is there any valuable troubleshooting information in it?

1. Halt and reboot.

RTE-A and RTE-6/VM are table- and list-driven. Rebooting will re-initialize these data structures. If the problem symptoms are affected by rebooting, those structures are suspect. Thus, we have refined our problem definition.

2. That's not supported.

There's probably a good reason why HP advises against a particular hardware or software configuration. You may have just found it. Check to be sure you're within tolerances.

3. That's fixed in the next revision.

You may have found a legitimate bug in HP's, another vendor's or your own code. Pull out a service request form if you suspect it's an HP bug, fill in the form and send it to one of the two addresses listed in the preface to your SSB.

4. Read the manual.

Did you misunderstand the proper procedure for configuring your system, or make an error in the use of a software utility? Check it against the documentation.

5. That's not a bug, that's a feature.

You may have an objection to the design of an HP product. Don't hesitate to file a service request asking that the design be changed, but recognize that other users may depend on the very design feature to which you object.

Conclusion

This paper has reviewed both a systematic approach to troubleshooting, and specific tips and tools that can be used to troubleshoot HP 1000 problems. The key to effective HP 1000 troubleshooting is the consistent problem analysis method. The tools and techniques described above can only provide information during the troubleshooting process. This information benefits the systematic troubleshooter, but the careless problem solver will become muddled in a morass of minutiae. With a consistent approach, problem solving becomes a stimulating challenge instead of an unavoidable burden.

Appendix

Problem Definition

Environment

Subsystems: _____

_____

_____

CPU: _____ O/S: _____

Peripherals: _____

_____

_____

Interface Cards: _____

_____

_____

Revisions: _____

_____

Symptoms

What: _____

_____

_____

_____

_____

_____

_____

_____

When: _____

Where: _____

Who: _____

Purpose

Intended Task: _____

History

Symptoms First Observed: _____

Changes: _____

What Has Been Attempted: _____

A.2 Useful Documentation

A.2.1 RTE-A

1. RTE-A System Design Manual, HP Manual Part Number 92077-90013

   This manual is designed to provide information that helps the user configure a system and troubleshoot system difficulties. It includes information on system data structures, system memory usage, and functions performed by system modules. The system tables chapter is especially informative.

2. RTE-A Quick Reference Guide, HP Manual Part Number 92077-90020

   This little gem of a manual extracts useful information from other RTE-A manuals and assembles it all into one handy guide. The tables and error messages sections are particularly convenient for troubleshooting.

3. HP 1000 A900 Computer Reference Manual, HP Manual Part Number 02139-90001
   HP 1000 A700 Computer Reference Manual, HP Manual Part Number 02137-90001
   HP 1000 A600/A600+ Computer Reference Manual, HP Manual Part Number 02156-90001

   These three reference manuals describe the hardware architecture and instruction sets of the A-Series CPU's. They are indispensible for system-level troubleshooting (for example, driver debugging). Valuable information includes instruction word formats and VCP commands, useful for troubleshooting with VCP and a logic analyzer.

4. RTE-A Driver Reference Manual, HP Manual Part Number 92077-90011

   This should be instantly accessible during any I/O troubleshooting. It provides information on proper configuration, correct format of control statements and legal uses of drivers and I/O cards.

5. RTE-A System Generation and Installation Manual, HP Manual Part Number 92077-90034

   Always check your gen. Always check your gen. Always check your gen. This manual and the RTE-A Driver Reference Manual

provide the information you will use to check your gen.

6. RTE-A Technical Specifications
   EXEC Control, HP Part Number 92077-90027
   I/O Control, HP Part Number 92077-90028
   Link Loader/Generator, HP Part Number 92077-90029
   BOOTEX/BUILD, HP Part Number 92077-90031
   Drivers, HP Part Number 92077-90032

   These documents are not very useful unless you also have access to source code. However, used in conjunction with the source code, they are extremely informative on the internals of the operating system and how programs and drivers interact with the operating system. I advise against using this information to tinker around with tables and lists, but a better understanding of precisely what is happening is obviously helpful for both troubleshooting and efficient design.

7. RTE-A Index and Glossary, HP Manual Part Number 92077-90036

   How to find out where almost everything is documented in the RTE-A manual set.

A.2.2 RTE-6/VM

1. HP 1000 M/E/F-Series Computers Technical Reference Handbook, HP Manual Part Number 5955-0282

   This manual is the M/E/F-Series equivalent of the A-Series Computer Reference Manuals. Chapter Two discusses the use of the front panel for examining registers.

2. RTE-6/VM Quick Reference Guide, HP Manual Part Number 92084-90003

   This is the RTE-6/VM equivalent of the RTE-A Quick Reference Guide. Keep one copy by your system console and one by your terminal.

3. RTE-6/VM System Manager's Reference Manual, HP Manual Part 92084-90009
   RTE-6/VM On-Line Generator Reference Manual, HP Manual Part 92084-90010

   It's even more important to check your gen under RTE-6/VM, since the generation procedure is more obscure than under RTE-A (although the generator is no longer much slower). These manuals and the appropriate driver manuals provide the

information necessary to do that checking.

4. Driver Manuals

   RTE-6/VM driver information is not all in one manual as it is for RTE-A. However, manuals are available for the various drivers.

5. RTE-6/VM Index to Operating System Manuals, HP Manual Part Number 92084-90001

   As the title suggests, this manual indexes all the RTE-6/VM operating system manuals.

A.2.3 Miscellaneous

1. HP Computer Users Documentation Index, HP Manual Part Number 5953-2460(D)

   This manual tells you the part numbers and titles of all sorts of interesting documentation, including those driver manuals mentioned above.

2. The Communicator/1000

   How do you tell if you have the correct revisions? You use your Communicator/1000 (formerly the Software Update Notice), or you look at the Software Numbering Catalog file. (The Communicator/1000 document is supplied as part of your update service. The Software Numbering Catalog file is included as one of the product update files which are supplied on magnetic tape or other media, again as part of your update service. The Software Numbering Catalog file is always named with the product number prefaced by an A. For example, RTE-A has the product number 92077, so the SNC file is A92077. Likewise, RTE-6/VM with product number 92084 has SNC file A92084.) You then list the relocatable file (for example, %FMGR) and compare the revision code in the relocatable with that in the Communicator/1000 Current Revisions chapter or the Software Numbering Catalog file. You can also tell what a module's revision codes are from gen listings and load maps.

3. The Software Status Bulletin

   Another document that comes as part of your update service, the Software Status Bulletin lists known bugs in HP 1000 products. You can compare your symptoms against the known bugs. The

Software Status Bulletin also describes workarounds where one is available, and fix information (the cause and the revision in which the bug will be fixed) when known.

A.3 Symbolic Debug/1000 Data Sheet

Symbolic Debug/1000

HEWLETT PACKARD

HP Program Development Software product number 92860A

Symbolic Debug/1000 is an interactive, symbolic debugger for source-level FORTRAN, Pascal, compiled BASIC, and Macro programs on RTE-6/VM and RTE-A based HP 1000 systems. Variables are displayed or modified using names from the original program. Load maps and program listings are not needed. One and two word integer, two, three and four word reals, logical, complex, character, Hollerith and most structured data types are supported. Symbolic Debug resides in a separate partition from the program being debugged to eliminate program code space intrusion. A single-stepping, source-line capability displays the current and adjacent lines during execution. Conditional breakpoints can be used to monitor variable values and stop the program at a specified value. Using the profiling capability, the user can determine which subroutine is using the most program time and optimize the code to decrease execution time. A small, simple command set, the use of dozens of English error messages, and an on-line "help" facility make Symbolic Debug/1000 a friendly and powerful programmer's productivity tool.

```
   40           RE   = DCOS  (ANG)
   41           IM   = DSIN  (ANG)
   42     2     IF  ( .NOT.  NEW  .AND.  K*KO  .GE.  1 ) GO TO 4
   43 C         ------------------------------------------------
   44 C         COMPUTE TWIDDLES IF NECESSARY ...
   45 C         ------------------------------------------------
   46           U(1) = DCMPLX( RE , -SIGN(IM,DBLE(K)) )
>  47           DO  3  I  =  2,L2N
   48     3     U(1)  =  U(I-1)*U(I-1)
   49           KO = K
   50 C         ------------------
   51 C         BUTTERFLIES.
   52 C         ------------------
   53     4     SBY2 = N
   54           DO 7 STAGE = 1,L2N

DEB.A> b 47/fft
Breakpoint set at 47/FFT
DEB.A> p
Break at 47/FFT
DEB.A> d L2N new re u(1)
L2N = 5      NEW = true      RE = 0.980785282244344
U(1) = (0.980785282244344,-0.195090312760225)
DEB.A> m L2N 6
L2N: 5 => 6
DEB.A>
```

Features

- **Interprets all code types and symbols used**
- **Can display source code during execution**
- **Non-intrusive — no Symbolic Debug code resides in user space**
- **Supports EMA and RTE segmentation**
- **Supports all simple and most structured data types**
- **Program profiler isolates slow subroutines**
- **Source line-by-line single stepping capability**
- **Up to 50 conditional breakpoints to stop program at specified variable value**

Functional description

Symbolic debug. Symbolic Debug recognizes the names, types, and locations of all of the variables and routines used in the program, eliminating the need for load maps, symbol table dumps, and mixed listings. The value of a variable can be examined as fast as its name can be typed.

Interactive debugging process. The user interacts with the program as it runs and can examine or alter variable values while the program runs without having to insert statements into the code. Bugs can be found fast since there's no need to recompile and load every time a new bug occurs.

Separate partition. Symbolic Debug resides in a 32-page memory partition separate from the program. No code space is lost and no extra statements are added in order to debug. The program being debugged runs exactly the same as it would normally. No bugs are introduced by the debugger, and more importantly, bugs don't disappear when the debugger is present, only to reappear when the debugger is not used. There is no need to restructure a program just to debug it.

Source-level symbolic. Symbolic Debug recognizes what line of source code is about to be executed, and identifies it on the CRT display. Programs can be debugged in the language in which they were written, without the need for inverse assemblies or mixed listings. There is no need to list files at all.

Detects RTE program violations. After a detected violation such as an attempt to access protected memory, memory locations can be examined to determine the cause of the problem. Symbolic Debug pinpoints the line of source code that caused the error, giving the operator an interactive tool for catching system violations.

Standard and conditional breakpoints. Up to 50 breakpoints enable Symbolic Debug to monitor program variables and halt program execution if a variable reaches a specified value. A large number of possible paths can be trapped and values can be quickly tracked through the program to determine where they go wrong.

Supports transfer files and message logging. Non-interactive debugging sessions may occur where users can submit debug commands in a file, and have results logged in another file. This automates the debugging process, so users don't have to wait for bugs whose symptoms may take hours or even days to occur.

Built-in profile monitor. Helps isolate slow parts of the program. High-level analysis of activity distribution within the program helps to identify time-consuming subroutines that should be optimized in order to improve execution time. For example:

Profile for program TEST

Routine	Amount	Histogram
OTEST	39%	**********
SUBR	27%	*******
OTES1	16%	*****
UTILITY	9%	***
OTES0	3%	*
Other (your code)	2%	*
Other (libraries)	3%	*

Profile for module OTEST: 39% of total time spent here

Line No.	Amount	Histogram
7	20%	*****
8	11%	***
9	33%	********
13	36%	**********

Debug command summary

B <Location>	Sets breakpoint at specified location.
C<Location>	Clears breakpoint at specified location
D <Locations>	Displays variable.
E	Aborts your program and exits Debug.
F/<string>	Finds string in source file.
G <Location>	Allows your program to proceed from specified location.
H	Displays histogram.
I <f1 [f2]>	Executes a set of commands from a file (f1) and optionally logs the output to f2.
L <Location>	Lists a screenful of source code in your program.
M <Loc> <val>	Modifies the value of variable.
O	Enters overview mode (enables profile monitor).
P <line>	Allows your program to proceed to the next breakpoint or specified line
S	Steps to the next line of source code.
T <Location>	Shows location executed without stopping program.
V <number>	Changes the number of source lines displayed on screen.
W	Shows callers of the current subroutine.
?	Help facility

Environment

Operating System

HP 1000 Computer System operating under RTE-A or RTE-6/VM, revision code 2226 or later.

Supported Program Languages

92836A FORTRAN 77, Macro/1000, Pascal/1000* and BASIC/1000C (Compiled code)*.

**Effective with A.84 revision.*

Ordering information

92860A Symbolic Debug/1000

92860A Symbolic Debug/1000, which must be ordered with Use Option 600, 700, or 890, includes:

1. Symbolic Debug/1000 software on one of Media Options 020-051.
2. Symbolic Debug/1000 User's Reference Manual (92860-90001)
3. Symbolic Debug/1000 Configuration Guide (92860-90002)

92860A Media Options

020: Software on 264x Minicartridges

022: Software on CS/80 cartridge tape.

041: Software on 1.2M byte flexible disc.

042: Software on 5-in. Minifloppy Discs.

044: Software on 3.5-in. Microfloppy discs.

050: Software on 800 bpi mag tape.

051: Software on 1600 bpi mag tape.

92860A/92860R Use Option

600: Use in A600+ or A600 system.

601: Upgrade from previous version of 92860A/R Opt 600 to latest version of same for customer NOT on support service.

700: Use in A700 or E/F-Series system.

701: Upgrade from previous version of 92860A/R Opt 700 to latest version of same for customer NOT on support service.

890: General license to use in A900 system or any other A/E/F-Series system, including right to purchase 92860R Opt 600/700/890 right to copy products for additional systems.

891: Upgrade from previous version of 92860A/R Opt 890 or 700 to latest version of 92860A/R for customer NOT on support service

92860R Right to Copy Symbolic Debug/1000 for Use on an Additional Computer System

The 92860R Right to Copy product, which must be ordered with Use Option 600, 700, or 890, is available only to customers who have previously purchased a 92860A product. 92860R consists of:

1. The license to make one copy of software purchased with 92860A for use on an additional computer

2 and 3. Same as for 92860A.

Software Support Products Available

See page 1-4 of Volume I of the HP 1000 Software Data book.

A.4 Bibliography

Adams, James L. (1974). *Conceptual Blockbusting, A Guide to Better Ideas.* W. H. Freeman and Co., San Francisco.

Reilly, William J. (1947). *The Twelve Rules for Straight Thinking, Applied to Business and Personal Problems.* Harper & Brothers, New York.

1034. FIXED/REMOVABLE DISK DRIVE PERFORMANCE ON THE HP1000

Craig Fix
Hewlett-Packard Co.
P.O. Box 39
Boise ID 83707

INTRODUCTION

Fixed/removable disc drives have played a major role in the area of mass storage for many HP1000 applications. Those products currently in use include the 7900, 7905, and 7906 disc models. On May 2, 1985, Hewlett-Packard introduced the 7907A. The 7907 is HP's newest fixed/removable disc drive featuring:

- CS/80 controller / HP-IB interface
- 20.5Mb fixed/20.5Mb removable formatted capacity
- 55 pound weight
- 1/5 the size of the 7906M
- Push button save/restore capability in two minutes
- Lower cost per Mb

Although the above features are highly attractive compared to it's MAC drive predecessors, how well does the 7907 compare to the 7906 in the area of performance?

This paper will focus on comparing the 7907 and 7906M in the areas of random and sequential access environments as well as backup on the HP1000 A900 and E-Series model computers.

REVIEW OF COMPONENTS AFFECTING PERFORMANCE

Before performance results are presented, it may be helpful to review some of the components affecting I/O performance of the 7907 and 7906M on the HP1000.

SOFTWARE TIME

Software time, the first level in an I/O call, includes the time to launch the request, and the time for the request to propagate through the I/O subsystem in RTE. For programs using EXEC reads and writes, this time is very small compared to total I/O time. This time will vary depending on CPU model, operating system, and disc driver being used.

DISC CONTROLLER TIME

The controllers used in the 7906 and 7907 are very different. The 7906M uses the 13037 multi-access controller (MAC), whereas, the 7907 uses a single board CS/80 controller. The MAC controller, a hardware intensive device has very little overhead, and can process commands very fast. The CS/80 controller has much of it's functionality migrated to firmware. This results in a more cost effective and more reliable disc drive, as well as providing significant improvements in I/O performance in multiple drive system configurations.

Table 1 compares the disc controller overhead associated with the 7907A and 7906M.

DISC MECHANISM TIME

The lowest level affecting I/O performance is the time required to seek to the target address and transfer the data. Seek time is primarily a function of the acceleration and deceleration capability of the actuator. Both the 7907A and 7906 have linear actuators. Their average random seek times are shown in Table 1.

Average latency, another component of disc time is equal to one-half the time to complete one revolution of the disc.

Transfer rate, the final component of disc time is affected by three items: (1) rotational latency, (2) density, (# sector/track), (3) and to a lesser degree the time required to transfer the data to the host. 7906M are shown in Table 1 below.

Table 1. Factors Affecting Performance

DISC MODEL	CONTROLLER TYPE	CONTROLLER OVERHEAD	AVERAGE RANDOM SEEK	AVERAGE ROTATIONAL DELAY	AVERAGE TIME TO TRANS. 1Kb	SECTORS PER TRACK
7907A	CS/80	4.0 ms	30 ms	8.5 ms	1.8 ms	64
7906M	MAC 16 bit//	.5 ms	25 ms	8.3 ms	1.1 ms	48
7906M Opt.102	MAC HP-IB	.5 ms	25 ms	8.3 ms	1.1 ms	48

PERFORMANCE TESTS

Performance tests were conducted with the 7907 and 7906 using the A-Series Model A900. Results for the E-Series 2109 with High Performance Memory were uncompiled at the time of submittal.

The following tests were conducted:

1) Random Access Performance
   Comparison of drive performance using randomly generated addresses for EXEC Read requests using 1K to 8K byte transfer lengths.

2) Sequential Access Performance
   Comparison of drive performance using sequential addressing and EXEC Read requests for .25K to 4K byte transfers.

3) Backup Performance
   Comparing how the 7907 and 7906 perform when copying the same set of files from the fixed disc to various backup devices using RTE utilities.

RANDOM ACCESS RESULTS

Figures 1 and 2 illustrate how the performance of the 7907 and 7906 compare in random access environments on the A900. In the single drive comparisons, the 7906M is only slightly faster than the 7907 at 1 Kilobyte transfers. Note in

Figure 1 that the difference diminishes slightly with increasing transfer size, since it takes less time for a sector to pass under the head on the 7907 than the 7906.

The 7907 can provide significant improvements in data throughput in A-Series system configurations designs where multiple disc mass storage solutions are employed. Note in both charts that the multiple 7907's per HP-IB improve aggregate data throughput, whereas additional 7906's per 13037 controller would yield no more throughput than one 7906. This improvement in multiple drive performance is possible due to the design of the CS/80 controller, and the I/O subsystem design in RTE-A. Using dedicated HP-IB interfaces for each disc can further improve I/O performance where multiple disc configurations are needed using the 7907 or 7906 Option 102.

SEQUENTIAL ACCESS RESULTS

Figures 3 and 4 represent the performance characteristics of the 7907 and 7906 under a sequential access environment. Here, the 7907 yields a five to ten percent improvement in performance through these transfer lengths. The performance edge of 7907 is attributed to the rate at which data can be read off the disc. Remember, 7907 has 64 sectors per track compared to 48 for the 7906. However, both drives have nearly the same rotational latency.

BACKUP PERFORMANCE

Often, the time to backup data on the disc drive is of great importance to the user. In addition, the availability of the disc during the backup may be just as important. The backup performance benchmarks were conducted using files under the new file (CI) system. All files were contiguous on the disc and accessed in the same sequence during backup. Both RTE disc to disc and disc/tape utilities were used.

A significant feature available on the 7907 is the ability to conduct a fixed removable disc backup in two minutes with the touch of a button. Restores of the removable disc to the fixed are possible in two minutes also. Backup performance for RTE-A are summarized in Table 2. Unfortunately, data for RTE-6 was uncompiled at the time of submittal. Clearly, the push button backup/restore provides the fastest backup possible and thus, minimizes system downtime. RTE-A disc to disc and disc/tape utilities perform nearly identically using the 7907 or 7906M Opt. 102.

Table 2 7907 VS. 7906M BACKUP PERFORMANCE

UTILITY	TO DEVICE	7907A	7906M Opt. 203
Push Button	Rem. Disc	11.71 Mb/min	Can't Do
COPYL	Rem. Disc	8.99 Mb/min	8.91 Mb/min
TF	7970E	1.24 Mb/min	1.37 Mb/min
TF	7974A	1.20 Mb/min	1.30 Mb/min
TF	9144A	1.02 Mb/min	1.04 Mb/min
ASAVE	7970E	3.70 Mb/min	3.68 Mb/min
ASAVE	7974A	8.32 Mb/min	8.14 Mb/min
ASAVE	9144A	1.97 Mb/min	1.92 Mb/min

SUMMARY

This paper has presented the performance results of the latest fixed/ removable disc drives from Hewlett-Packard. For A-Series systems, the 7907 provides comparable performance to the 7906 in random and sequential access I/O bound environments. The 7907 can yield even more significant gains in performance in multiple spindle configurations with other 7907's or CS/80 fixed disc drives. In addition, the 7907 provides this level of performance while at the same time providing more capacity, lower cost, and less maintenance than the 7906. Although the results for RTE-6 were not presented here, they will be available at the time this paper is presented.

7907 and 7906M Performance for Single/Multiple Drives

Random EXEC Reads Using an A900

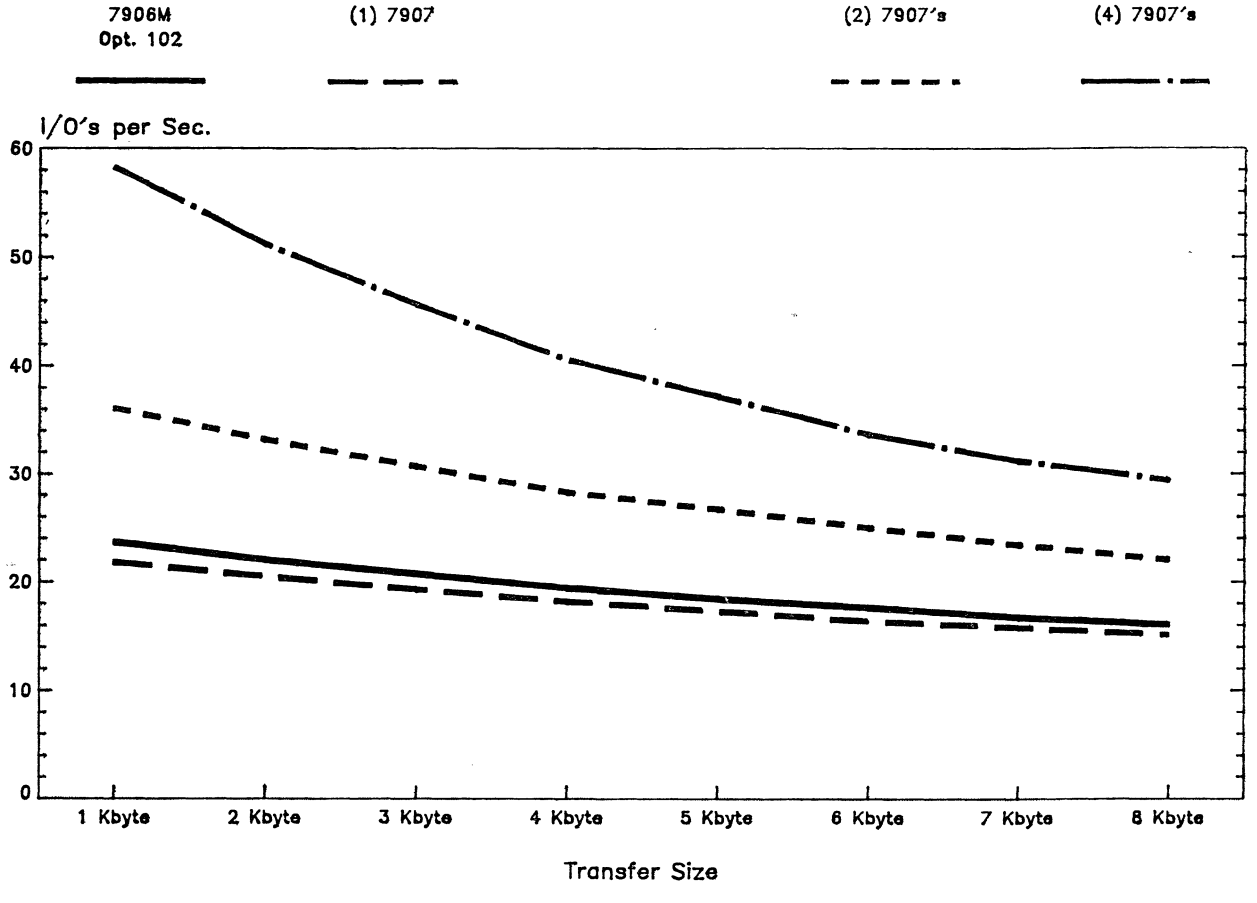


Figure 1

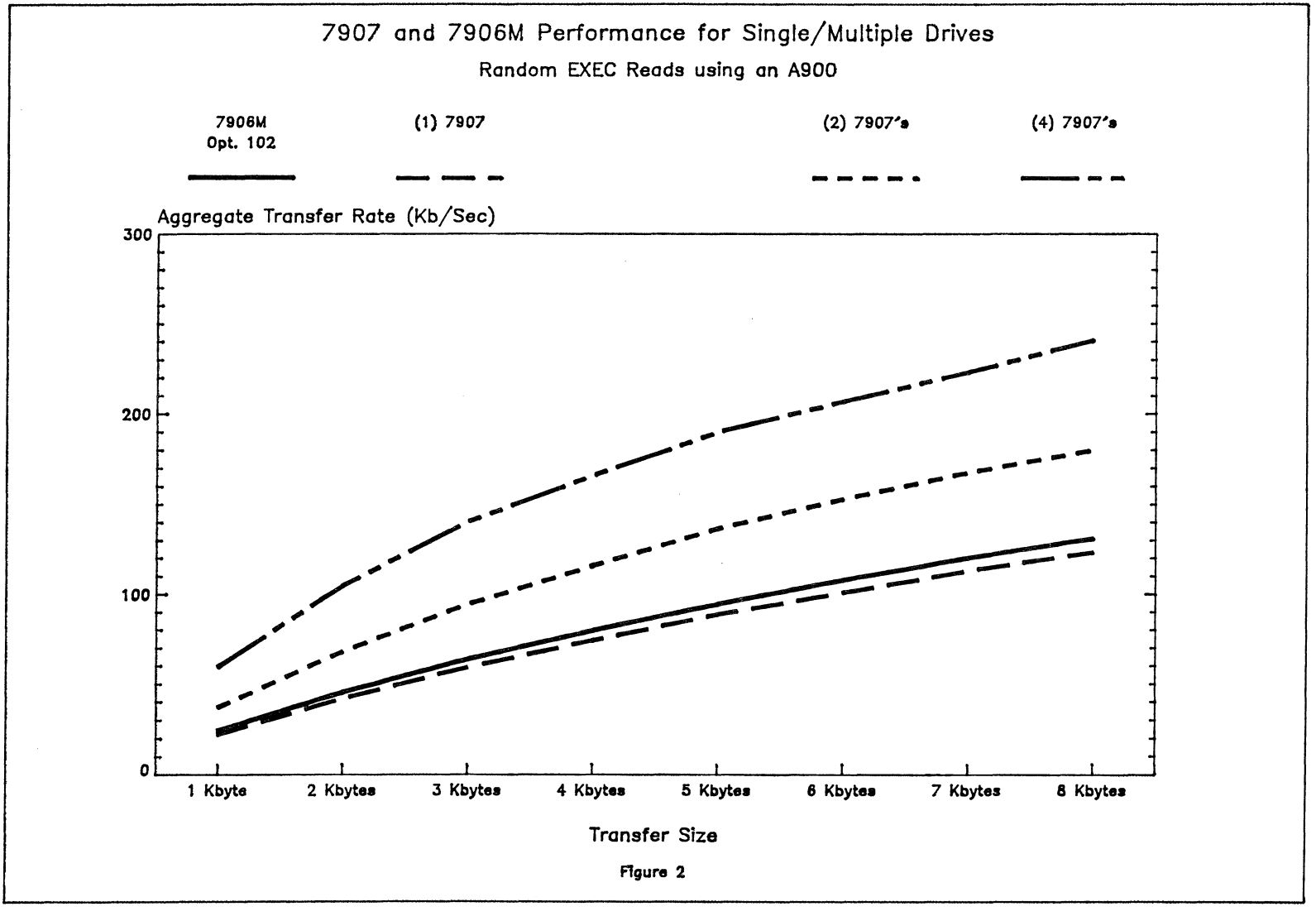

7907 and 7906M Performance for Single/Multiple Drives
Random EXEC Reads using an A900
7906M
Opt. 102
(1) 7907
(2) 7907's
(4) 7907's
Aggregate Transfer Rate (Kb/Sec)
300
200
100
0
1 Kbyte
2 Kbytes
3 Kbytes
4 Kbytes
5 Kbytes
6 Kbytes
7 Kbytes
8 Kbytes
Transfer Size

Figure 2

Sequential Access Performance Results for the 7907 and 7906M

Various Transfer Lengths using EXEC Reads on the A900

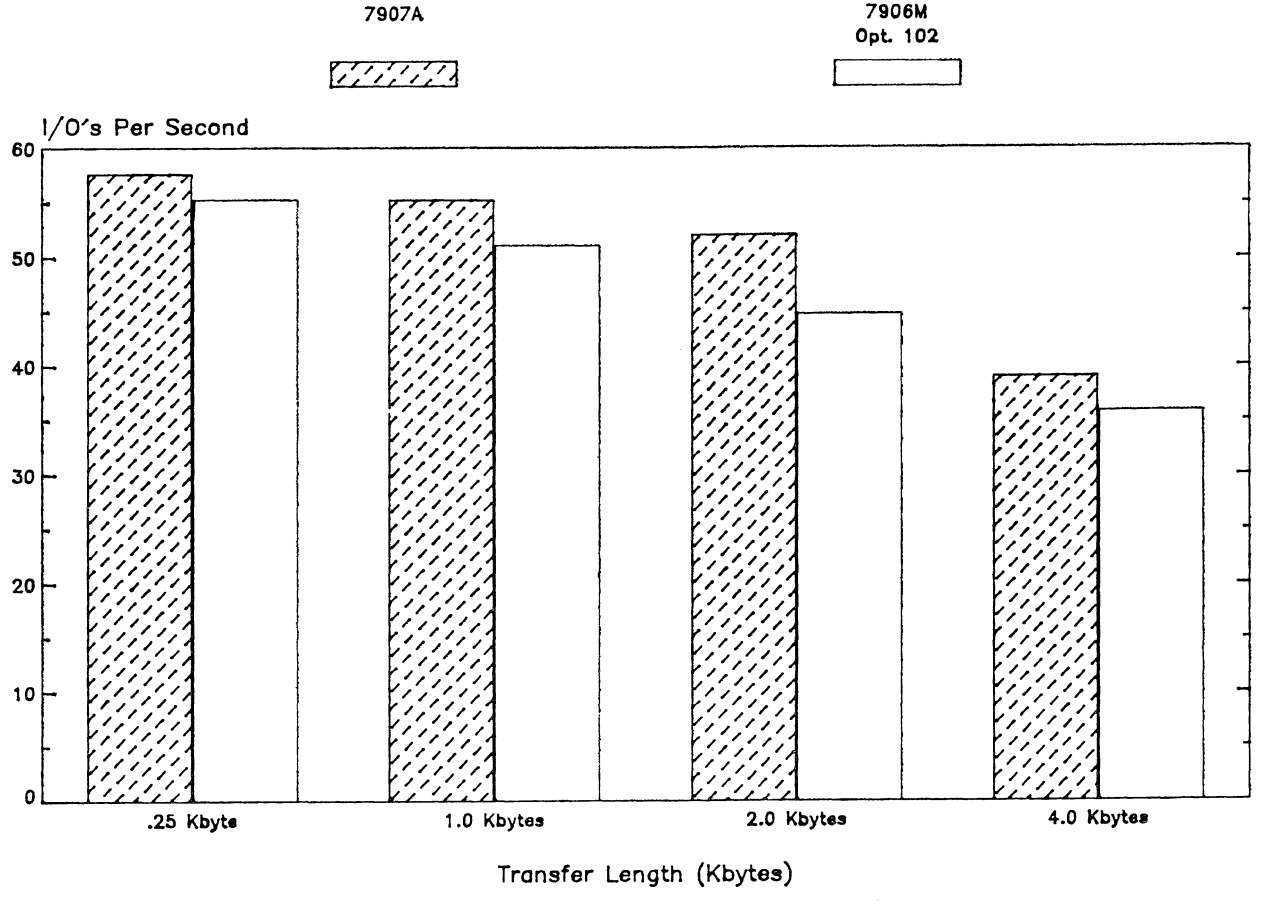


Figure 3

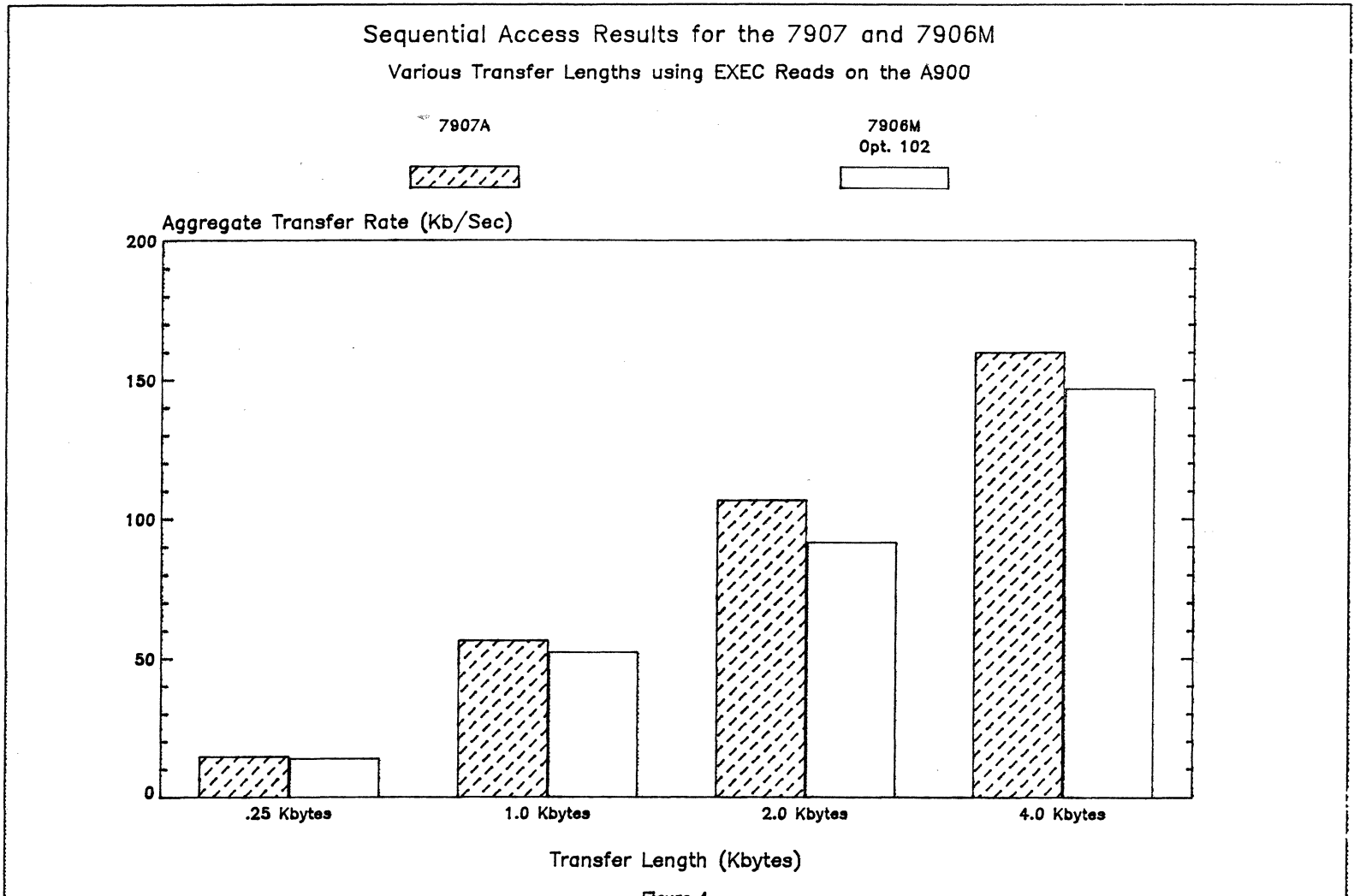

Sequential Access Results for the 7907 and 7906M
Various Transfer Lengths using EXEC Reads on the A900
7907A
7906M
Opt. 102
Aggregate Transfer Rate (Kb/Sec)
200
150
100
50
0
.25 Kbytes
1.0 Kbytes
2.0 Kbytes
4.0 Kbytes
Transfer Length (Kbytes)

Figure 4

1035. BUILDING AN IN-HOUSE TIME SHARE SERVICE CENTER

R. Arthur Gentry
AT&T Communications
811 Main Street
Kansas City, MO 64141

This paper will describe how the Midwest Region of AT&T Communications set up an in-house time share system for use in their Regional Headquarters building. The system currently has 5 major applications running, being used by 5 departments. The paper will describe each application as well as overall system design and operation.

The System

The system currently consists of 3 F Series processors, 422mbytes of disc storage, 2 800bpi tape drives, 1 1600bpi tape drive and 1 tri- density tape drive. Each processor is equipped with 2mbytes of high performance fault control memory, up to three 8 channel multiplexors and 1 I/O extender. (See figure 1) All three processors are linked together using DS/1000 IV over a X.25 network, hardwired. The system also has a 2608S high speed line printer for shared use. All terminals are hardwired to the system either through EIA line extenders, direct connection or 9600bps data modems over dedicated lines.

The processors are divided by major applications, with all users having remote logon capability to get to any application. Processor #1 has a CAD package on it only. It is a very disc intensive application and studies are currently under way for the installation of a solid state disc for this. Processor #2 has all our major database applications and processor #3 is our miscellaneous applications.

Each group of users has its own terminals, printers, plotters, etc. and through the use of remote mapping, each appears to be connected to which ever processor the user is currently logged into. Processor #3 is set up to allow restoral of processors #1 or #2 in the event of a system failure, as they have our most critical operations on them.

Transmission Engineering

This group is in a remote location with two HP150s and several HP-IB devices working over a 4 channel statistical multiplexer on a 9600bps data modem. The HP-IB devices work over a 37201A HP-IB to RS232 extender and include a 9872C plotter, a 9111A digitizer and a 2225A printer. They have applications running on two of the processors. Their main application, on processor #3 is an in-house designed project management package. It is used to write project sheets for the design and installation of communication lines to our various remote relay stations. They also access processor #1 for the CAD package to make drawings of the layouts.

Graphics

This group is made up of mechanical draftsmen and graphics artists. They are

responsible for all the graphics arts projects for the region as well as a large percentage of the mechanical and electrical drawings for our various engineering departments. The draftsmen are using the Holguin CEADS/CADD 2000 package for all their mechanical and electrical drawings. The graphics artists are using GRAFIT/1000 for charts and posters used by various management, marketing and public relations personnel. They currently have six 2627A terminals e/w 17623A tablets, one 2648, connected via either an 8 channel multiplexor, EIA line extenders, or direct cabled; two 7475B plotters, one 7585B plotter, one 9872C plotter and one 2632A printer all connected via two 37203A HP-IB extenders and one 44" X 60" CalComp digitizer connected via the multiplexor. The users in this group access processor #1 for the CAD package and processor #3 for the GRAFIT package.

Construction Engineering

This group is responsible for the design and construction of our various remote relay buildings. They have two HP150s and a 2627A terminal connected via EIA line drivers and a 7475B plotter, 9111A digitizer and a 2632A printer connected via a 37203A HP-IB extender. They access processor #1 for the CAD package for their mechanical and electrical drawings, and processor #2 for their database of remote buildings. This database contains all the information on each site, including; when built, how much, continuing costs, modifications, inspection reports, drawing records and accounting information.

Medical

Approximately two years ago, the company started an in-house fitness program for its employees. This program proved to be so popular, it generated a tremendous amount of paper work for the center staff. A database and analysis system was designed to help elevate some of this work. The database contains all the pertinent medical information for each participant, their prescribed workouts, and an automatic method for each user to receive their workout information and store results back into the database for later analysis. They have 1 150PC e/w a 2225A printer for use by the center personnel and a 150 terminal e/w a thermal printer for use by the center participants.

Real Estate Engineering

This department is responsible for facilities planning for our various administrative offices in the region, including floor space planning for our new headquarters building, just now being built. Using the Holguin package, which has a built in "Bill-of-Materials" package, they are able to generate a list of how many desks, partitions, files, electrical outlets, phones, etc. that are needed for each floor. Also, using the "layering" concept, they are able to generate plots of 1) the "raw" floor, 2) electrical only, 3) communications only, 4) mechanical only, or 5) any combination of these.

Future Directions

One of the major communications enhancements to be installed in our new building, is a LAN. It is our intention to interface this system via that LAN to all terminals not requiring graphics. The reason for not including graphics capabilities is, it is felt that with current LAN technologies, the graphics

traffic would totally overload the network. We also plan on bringing our R&D lab on line to do their design drawings on the system. One of our major goals, is to bring all our regional drawings, approximately 20,000, on line and to place numerous terminals in all of the engineering departments, so any engineer can have instant access to the latest version of any drawing, and also have the ability to do "source level" modifications to those drawings. Currently all drawings are "penciled" by the engineer and sent to the graphics department for the actual master drawing changes. Another major undertaking will be to interface our WANG word processing center through the 1000 to a photo typesetter for high quality printing.

We anticipate the system growing to over 100 terminals accessing over 30 major applications within the next 2 years.

THE APPLICATIONS

Computer Aided Design & Drafting

We are using a 3rd party software package called CEADS/CADD 2000 by Holguin and Associates of El Paso, Texas. After much searching for a package that would do the type of operations needed, we found that this package had the best price/performance of any we found useful. This package has many outstanding features including, layering, ease of use, short training time, automatic dimensioning, good response time, and a "bill of materials" package. Our only major problem has been disc contention. The package does extensive disc I/O and with 15 users all doing CAD work simultaneously, the disc gets thrashed very heavily. We are currently studying the possibility of installing a solid state disc for the package work files. Investigation shows that we can expect a 30% improvement in system response by not using an electo-mechanical disc.

Our drawing library currently contains over 3000 drawings and is growing at a rate of 10 to 15 per day.

Business Graphics

For our business graphics, charts, announcements, etc. we were using TELEGRAF on a time-shared main frame over 1200bps dial-up modems. Since we were running up $1500.00 per month in time-share charges, we decided to investigate moving our work over to the 1000. We found the GRAFIT/1000 by Graphicus of Santa Clara, California, is able to replace almost all of our current TELEGRAF use. It is very friendly to use, and allows the user to "see what you've done" as he goes. That, along with the ability to run at 9600bps made the loss of those few TELEGRAF features very painless. Also, the software writers have been excellent in incorporating our suggestions into their package, so that we expect GRAFIT to totally replace TELEGRAF in the very near future. As of this time, we have totally cut the strings to TELEGRAF, and were able to transfer most of our TELEGRAF command files over to the 1000 for use by GRAFIT with very few generic modifications to the commands.

Project Management

Our Transmission Engineering group is responsible for designing and implementing projects to install communications equipment at our various remote repeater

locations. This was a totally manual operation, with the engineer drawing the circuit layout and writing up the ordering information for the actual equipment. We have now totally mechanized this, with a tremendous reduction in the amount of time it now takes the engineers to get a project out. The circuit drawings are now done on the Holguin CEADS/CADD package, and the order writing is handled by an in-house developed project writer. The engineer now needs only to "fill-in-the-blanks" on the CRT screen, and the system takes care of producing the actual project letters. This information is databased for easy retrieval and modification if necessary.

Fitness Center

The medical department needed a method of keeping track of who was using the company fitness center and how much their overall health improved by use of the center. We developed a totally mechanized package which allows the center users to "log-in", receive their personal workout prescription, and when leaving, log their progress. This data is then analyzed by the medical staff for trends in employee wellness.

When a user enters the center, they use a HP150 Touchscreen terminal to enter their Social Security number, after the system confirms the identification, it prints via the built-in thermal printer the users workout list. When logging out of the center, the user enters various information, telling the system what he/she did that day. The Touchscreen works very well, as the users are not dripping sweat and grime into the keyboard.

The center staff has application programs that allow listing of current users, analysis of center usage, input of user base and testing data, maintenance of locker assignments, print individual fitness profiles, review workout cards that have been flagged by the system as abnormal, and analysis of health related data.

BUILDING THE SYSTEM

All the major system components, CPU's, disc drives, tape drives, some of the terminals, and printers, were "salvaged" from other company locations that had purchased them and no longer needed them. Several other items, such as printers, disc controllers and terminals were purchased on the used equipment market. The remainder, CPU upgrade kits, multiplexors, additional terminals and plotters, were purchased new from HP. This made our initial investment in the system very low. For 3 CPU's, 5 disc drives, 4 tape drives, numerous terminals, printers and plotters our total hardware investment was less than $50,000.

All software was purchased "new" from HP or other 3rd party vendors, and with carefull system planning, the only software we needed right-to-copy on is the operating system and the DS/1000 IV package. This helped keep the software costs low as well. Our total investment in software was approximately $100,000, with half of that being spent on the CAD package.

Another item that has helped keep the system operating costs low is hardware maintenance contracts. We elected to "split" our various pieces into criticality groups and write contracts accordingly. The items the system can't live without, CPU's and disc drives, were put on a standard 4 hour response. Important, but not critical items such as plotters and the system printers, were put on next

day contracts; and the remainder such as terminals, small printers and plotters were put on a once per week contract. This arrangement saved us several thousands of dollars over putting everything on the standard same day service contract.

All terminals are interfaced directly to the processors via either direct RS232 cables, EIA line drivers or 9600bps data modems. We have found, by using good quality shielded cables, we can extend the RS232 lines to 250 feet with little problem. You must, however, be very carefull about running the cables over or near any AC power lines, as noisy power can easily introduce noise into the cables. For distances over 250 feet but less than 2000, EIA line drivers have worked very well. They are totally transparent and do not hinder line rates at all. Where there is a cluster of terminals, an EIA line driver multiplexor is a very economical choice, up to 8 9600bps terminals or devices can share one EIA 4 wire cable, with no noticeable degradation in performance. For remote terminals over 2000 feet, 9600kbs DDS modems are the choice and again, where there is a cluster, a multi channel multiplexor is quite a bit cheaper than several data modems. Be very carefull about selecting terminal line speeds, as too slow a line, particularly when doing graphics intensive work, can cause your clients to spend more time waiting for the terminal to finish than doing actual work.

Most of our plotters, printers and digitizers are interfaced via HP-IB. There are several advantages to this; moving devices from one area to another as well as from the HP1000 to a desk top computer is very easy, several devices can share the same interface and the devices are easy to find on the used equipment market. However, two items to consider are; how many and type of devices to place on one interface and the cabling restrictions enforced by IEEE-488. One must be very carefull, particularly when interfacing plotters, not to put too many devices on any given interface, one 7585 plotter can "hog" the line very easily. When dealing with cabling restrictions, HP-IB extenders work extremely well and allow you to extend the HB-IB line up to 3000 feet using coaxial line. You must, however, balance the cost of the extenders, $2040.00 per pair, versus using RS232 interfaces. When working with remote HP-IB devices over a modem, the RS232 version of the HP-IB extender makes very good sense, as it also functions as a built-in multiplexor.

When placing applications on the system, one very important consideration is disc usage. Be very carefull about placing several highly disc intensive applications on the same disc; even better, if possible, divide applications over multiple disc controllers. Traditionally, disc contention is the biggest bottle neck in any system, and particularly in a time share operation, this can become a very serious problem, if users are constantly waiting on the disc. Disc controllers for CS-80 type discs are fairly inexpensive, $800.00 to $1000.00 and carefull shopping for MAC controllers can sometimes reveal a deal on them. We found a used MAC controller for $600.00 !!

On system operation, there are many things that can contribute to a well run and reliable operation for your clients. Schedule PMs, system generation switches and off-line backups for off hours. There is nothing a client hates more than being in the middle of a "hot" project only to be told he must get off so "system work" can be done. As for backups, we highly recommend requiring all applications to use the new CI file package. Among its many advantages, is the ability to do incremental backups of all accessed files. We keep all our file

backups back to day 1, and if you really want to impress a client, wait until he/she asks if a file created 2 years ago can be restored and you tell them "no problem"!! Our current backup policy is, incremental backups every night, disc backups (PSAVE) every Friday and IMAGE/1000 II database backups every weekend (done remote from the system managers home). PSAVE and IMAGE backup tapes are rotated every month, so we have the ability to restore the system backwards up to 5 weeks ago. I might also note that we have converted all our IMAGE applications to take advantage of transaction logging, so we also get a dynamic backup of all IMAGE database work done between weekly backups.

As well as backing up the data, you must also backup the system manager, particularly if it is a one man operation, such as ours. This can be as simple as having a "key operator" who is familiar with system operation and knows where to call for major help or a second person working directly with the system manager. One serious mistake I have seen in several operations has been the one "key" person with a large amount of system operation information "in his head" and no one working with him to share this information with. Systems have spent days of down time because the key person was unavailable and no one else knew his "secrets".

Training is another very large consideration when going into this type of operation. If you are going to make the investment in hardware and software, you must also make the investment in proper manager and operator training to get the most return on your investment. In our system, the system manager is responsible for getting training for himself on all software used on the system and then train the individual system users. Hewlett Packard as well as some other vendors, have created several self pace in-house training packages that make in-house training very economical and easy. We have found, however, that some very large applications, such as CAD, have required that the individual users go to the vendors training to get the most out of the package. Training can be expensive, but if you truely want the most out of your hardware/software bucks, it is a must.

One last note, when picking your system manager, he/she must have the following qualities; 1) have or is willing to get detailed system operation knowledge, 2) a self starter, 3) willing to work closely with system users, 4) willing to spend that extra time to find the best value for the dollar, and possibly most important 5) willing to go out into your business and seek new applications to add to the machine. If the operation is allowed to stay status quo, the system will not be giving you additional time and work savings and therefore no new return on investment. Your system must, at minimum, "pay for itself", or it is a total waste of company resources.

Conclusions

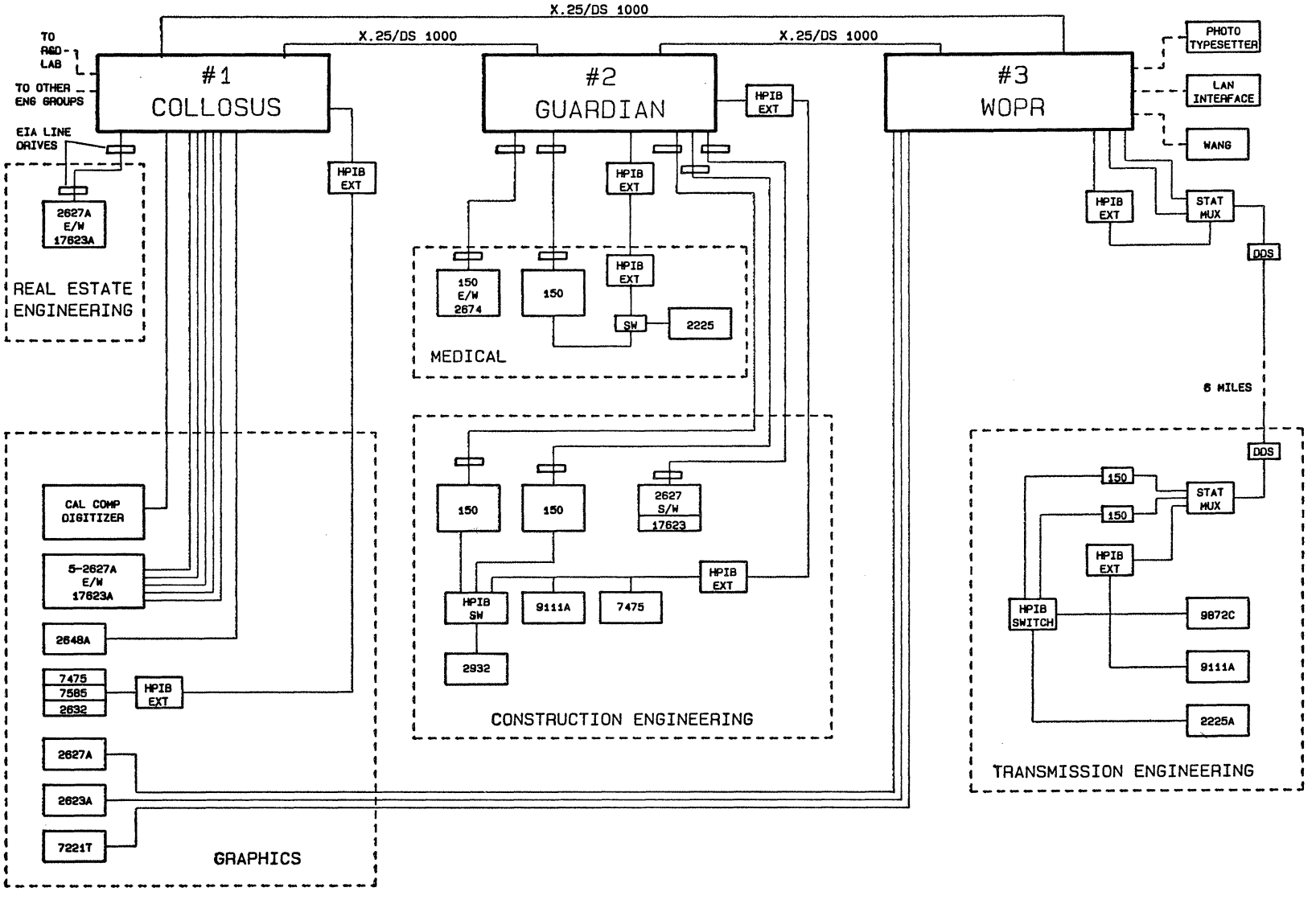
This has been a very inexpensive project, most of our hardware was second hand, either bought on the used equipment market, or salvaged from other departments, where it was no longer needed. Our major outlay has been for 3rd party software packages. Currently only one person runs the entire operation, so payroll expense is very low, compared to the operations the system is supporting.

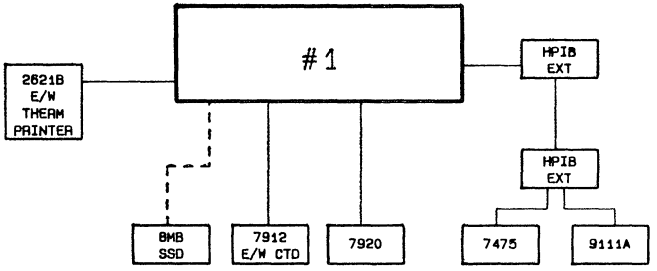
Some very critical observations; with a very limited support staff, one person, support services from all software vendors is absolutely critical to the continued

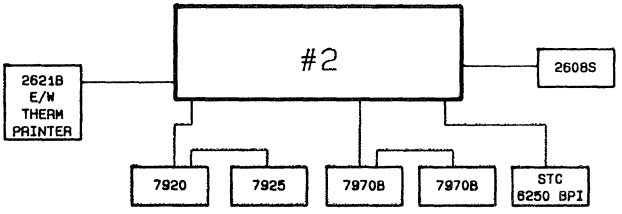
success of an operation of this type. The size of the staff does not allow the luxury of digging out problems locally, the system manager must have expert help he can call on to assist in application problem resolution quickly. The major item to remember, you must keep your client base satisfied to the best of your ability, or they will go back to their old methods of operation.

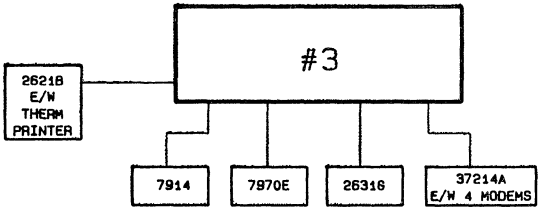
All maintenance costs, hardware and software, are prorated back to all the users of the system. This keeps our departmental expense very low. We are only responsible for acquiring system hardware and initial software expenses. Individual users are responsible for acquiring their own terminals, printers, plotters, etc. Individual or department application software development by the system manager is provided on a "time and materials" basis, and system applications for general use are provided "free".

In conclusion, we have found by carefully keeping a handle on your expenses, shopping for the best hardware values, and creative maintenance contracts, and user charge backs, a local time share operation is a very good use of company resources, with surprisingly high paybacks and savings. Hewlett Packard and other software vendors continue to come out with more new and exciting software packages to make this type of project more appealing.


X.25/DS 1000
X.25/DS 1000
X.25/DS 1000
TO R&D LAB
TO OTHER ENG GROUPS
#1
COLLOSUS
#2
GUARDIAN
#3
WOPR
PHOTO TYPESETTER
LAN INTERFACE
WANG
EIA LINE DRIVES
HPIB EXT
HPIB EXT
HPIB EXT
HPIB EXT
STAT MUX
DDS
2627A E/W 17623A
REAL ESTATE ENGINEERING
150 E/W 2674
150
HPIB EXT
SW
2225
MEDICAL
6 MILES
DDS
CAL COMP DIGITIZER
5-2627A E/W 17623A
2648A
7475
7585
2632
HPIB EXT
150
150
2627 S/W 17623
HPIB EXT
HPIB SW
9111A
7475
2932
CONSTRUCTION ENGINEERING
150
150
STAT MUX
HPIB EXT
HPIB SWITCH
9872C
9111A
2225A
TRANSMISSION ENGINEERING
2627A
2623A
7221T
GRAPHICS


#1
2621B
E/W
THERM
PRINTER
HPIB
EXT
HPIB
EXT
8MB
SSD
7912
E/W CTD
7920
7475
9111A


#2
2621B
E/W
THERM
PRINTER
2608S
7920
7925
7970B
7970B
STC
6250 BPI


#3
2621B
E/W
THERM
PRINTER
7914
7970E
26316
37214A
E/W 4 MODEMS

9001. HP-UX- HEWLETT-PACKARD'S STANDARD UNIX OPERATING SYSTEM

Judy Guist
Hewlett Packard Co.
Information Technology Group
19447 Pruneridge Ave.
Cupertino CA 95014

ABSTRACT

HP-UX is Hewlett-Packard's standard UNIX operating system. There are three major goals guiding this standard. They are: 1) - effortless migration of applications between any system in the HP-UX family, 2) - easy migration of applications from other popular UNIX environments, and 3) - add value without impairing ease of migration. This paper will describe how that standard was originally created, what it is based on, and how it is maintained. Included will also be a description about how conformance is checked as well as the standard's current direction and growth. In addition, a comparison to other popular UNIX systems will be presented.

9002. HP-UX FOR I/O CONTROL

Robert J. Schneider
Hewlett-Packard
Fort Collins System Division
3404 East Harmony Road
Fort Collins, Colorado 80525

I. Introduction

Historically, Hewlett-Packard's Fort Collins System Division (FSD) has produced single user workstations that served as excellent I/O controllers. There are numerous examples including the series 200 Pascal workstation, the series 200 Basic workstation, and the series 500 Basic workstation. These workstations had extensive features to allow users to control instrumentation systems on HP's IEEE 488 bus (HPIB), 16 bit parallel (GPIO), and RS-232 serial interface cards. In some cases other less common interface cards were supported as well.

When FSD introduced their multi-user HPUX workstations there were no provisions for instrumentation I/O capabilities. At that time it was thought that the HPUX workstations would be used as development workstations and there would be little interest in instrumentation I/O capabilities. This assumption proved to be incorrect. Realizing that there was a large demand for these capabilities the decision was made to add device I/O capabilities for the HPUX products.

II. HP-UX Device I/O Development

The development of a device I/O library (DIL) for HPUX began with forming a committee of I/O oriented people from several of HP's divisions with HPUX products. The purpose of the committee was to design and agree upon a device I/O standard to be implemented on all HPUX machines. The highest priority goal was to provide a compatible device I/O library on all HPUX workstations so that user code could be portable from one workstation to another. This goal caused some deviations from how I/O functionality was typically presented to the user in past HP products.

HPIB capabilities were given the highest development priority followed by GPIO capabilities. Some RS-232 capabilities currently exist in HPUX through the terminal driver and terminfo, therefore there has been little effort for RS232 interfaces.

It was determined that the cleanest method for achieving the code portability goal was to provide a library level for device I/O support. This would allow the various implementors to add the functionality as they saw fit for their environment. The library would serve as a layer to hide as many system dependencies as possible.

Initially, DIL was designed to provide a low level building block interface to the user. The intention was to provide all the necessary functionality but at a low level. It was felt that general routines could be built on top of the functionality provided. The initial feedback to this concept was unfavorable. This provided the motivation for some initial changes in the user interface.

Some of the changes to improve the user interface include auto-addressing for HPIB I/O transactions. In HPUX there are device files that refer to specific devices. These device files typically contain information associated with that device. In the case of HPIB device files addressing information is stored as well. When the user opens the device file, the operating system stores this addressing information for later use. Any read or write operation on this opened device file will have the addressing sequences supplied automatically by the operating system.

Another addition was the inclusion of the hpib_io function. This is a function that allows the user to build up a series of I/O transactions in a data structure and then pass them into the system via the hpib_io intrinsic. This call will perform all the requested transactions. Only one system call is supplied by the user to perform the series of I/O transactions. This can be very convenient when a given protocol is known ahead of time. It is important to note that this function adds no new functionality.

DIL will officially be supported under the C, FORTRAN, and Pascal languages. Currently, FORTRAN and Pascal impose a slightly different interface to the user. There are plans to resolve this issue in the future.

After the device I/O committee agreed on the proposed device I/O library functionality it was presented to HP's Technical Working Group to verify that it would fit into HPUX properly. After several minor revisions the device I/O library became part of the HPUX standard.

As more products support the device I/O library there will be a need for a validation suite to determine adherence to the standard. Currently there has been some work done to ensure that the mainline functionality works properly. However, much more extensive validation testing is required for the future.

III. Overview of Usage

The user is provided with several sets of functions. The first set is generic and applies to both GPIO and HPIB interfaces. The following functions fall into this class:

io_eol_ctl	set read termination character sequence
io_get_term_reason	determine termination cause on last read
io_interrupt_ctl	enable/disable interrupt handlers
io_on_interrupt	set up interrupt handler
io_reset	reset specified interface card
io_speed_ctl	specify desired I/O throughput rate
io_timeout_ctl	establish time limit for I/O operations
io_width_ctl	set width of data path

This next set of functions applies to HPIB interfaces only:

hpib_abort	stop activity on specified HPIB
hpib_bus_status	return status of HPIB interface
hpib_card_ppoll_resp	configure parallel poll response
hpib_eoi_ctl	control EOI assertion on writes

hpib_io	vectored reads and writes
hpib_pass_ctl	change active controllers
hpib_ppoll	conduct a parallel poll
hpib_ppoll_resp_ctl	control response to parallel polls
hpib_ren_ctl	control the remote enable line
hpib_rqst_srvce	request service from controller
hpib_send_cmnd	send ATN data (commands)
hpib_spoll	conduct a serial poll
hpib_status_wait	wait until status item becomes true
hpib_wait_on_ppoll	wait for parallel poll response

The last set of functions applies to GPIO interface cards only:

gpio_get_status	return state of status lines
gpio_set_ctl	set control lines

The read, write, open, and close functionality is provided by the standard HPUX read, write, open, and close intrinsics. This unified I/O approach allows the nice features of HPUX I/O to remain, for example, I/O redirection.

One of the most obvious deviations from past I/O functionality is the absence of functions to generically read/write interface card registers. Since interface card registers are very hardware specific, and not portable, this functionality was replaced with status and control functions, for example hpib_bus_status, that return or set information of interest. This method allows the hardware dependent interface registers to be hidden at the library level.

The interrupt mechanism differs from the past as well. The HPUX I/O interrupt mechanism works in a manner similar to the HPUX signal mechanism. The user can set up different interrupt handlers (procedures) for each uniquely opened select code. The user can even specify different handlers for the same select code by multiply opening the same select code. An example of the advantages that this scheme offers would be that the user could set up different interrupt handlers for different parallel poll responses.

Another noteworthy function is the io_speed_ctl call. This call was added to help machines with limited DMA capabilities. On these machines the user can specify a desired I/O rate, within the limits of the machine, to allow the machine to determine an appropriate method of transfer. On the series 200 HPUX system for example, if a requested speed can be satisfied without DMA then another method will be chosen should no DMA be immediately available. The series 500 always provides DMA and therefore returns without doing anything on this call.

In general, most of the I/O functions in HPUX have direct mappings with I/O functions from past products. Therefore, conversions of I/O application programs on past workstations should be relatively straight forward.

At this point a typical example might be in order. The following example waits for a device to request service. When service has been requested, a serial poll is performed to verify that the desired device was in fact the one requesting service. A data byte is then read.

```
#include <dvio.h>
#include <fcntl.h>
#include <stdio.h>
main()
{
     int eid, atof();
     extern int errno;
     double data;

     eid = open("/dev/hpib",O_RDWR);
     io_timeout(fd,1000000); /* set up 1 second timeout */
     if( hpib_status_wait(eid,SRQ) == -1)
             if(errno == 5)
                     printf("error, possible timeout0);
             else
                     printf("hpib_status_wait failed0);
     else
     {
             if( hpib_spoll(fd,my_dev_address) & 64 ){
                     hpib_send_cmnd(fd,address_info,4);
                     read(fd, bytes, 8);
                     data = atof(bytes);
                     printf(data = %lf0data);
             }
     }
}
```

This example illustrates how the DIL functionality is presented to the user as a series of library level calls. All DIL calls are function calls and therefore have return values associated with them. In many cases the return value is only used as a success/fail indicator. In other cases, for example hpib_spoll, the return value may be a parameter of more interest.

Note the usage of the HPUX atof function to convert the character data to floating point format. HPUX offers atof, atoi, sscanf, and sprintf for conversion utilities. atof and atoi are specialized high performance subsets of sprintf.

Other usage considerations include additional functionality for performance enhancers. Better and more predictable performance can be attained by using the HPUX real time enhancements along with DIL. A typical example might be to use the plock and rtprio intrinsics to prevent a process from being swapped out and to put it in the real time priority range. A common place to employ these intrinsics would be in conjunction with the interrupt handler. This allows users to cope with situations where the time to swap a handler into memory might be intolerable.

The series 200 has provided an extension to DIL for extreme performance improvements most notably in the small data transfer range. The series 200 HPUX implementation has a call to provide memory mapped I/O for the user. Since many instruments transfer data in small quantities, for example 8 bytes is typical for a digital voltmeter, this can be very useful.

The series 500 provides memory locking intrinsics that allow a user to perform I/O directly from user memory. Normally, data is transferred from the user space to the system space. By performing I/O directly from the user space a significant performance improvement can be obtained on data transfer rates.

IV. Performance Observations

Since the HPUX systems are multi-user and multi-tasking systems, some additional information must be supplied to understand the performance conclusions. Performance figures discussed in this paper refer only to lightly loaded systems with one active user. Performance could vary drastically with multiple users and multiple tasks executing in parallel with I/O performance benchmarks.

Performance results have been encouraging. The series 500 HP-UX I/O rates are very similar to the series 500 Basic I/O rates. In fact, the HP-UX system is capable of outperforming the Basic system on the Nelson benchmark by a small percentage.

The initial results on the series 200 HP-UX system were mildly discouraging when compared to the series 200 Basic system. As many people know, the series 200 Basic system was tuned to perform quick I/O data transfers, especially the small size transfers typical of instruments. However, the developing engineer for series 200 DIL implemented a memory mapped I/O scheme that has amazing I/O performance. Initial results show that for small transfers, the series 200 HPUX memory mapped I/O scheme is capable of producing I/O rates approximately six times faster than the series 200 Basic system.

The hpib_io routine may also be used for performance improvements. The actual performance improvement varies with the number of I/O transactions that are bundled into the one call; however, on the series 500, request blocks with around 7 transactions requested can typically enjoy about a 15% improvement over making 7 separate I/O calls.

Finally, auto-addressing allows users to cut the number of I/O requests in half. Without auto-addressing the user is required to perform addressing manually with the bus command to send ATN data. By allowing the system to address the bus significant performance improvements can be observed.

V. Conclusions

The conclusion reached was that device I/O fits nicely into the HPUX scheme. Acceptable levels of functionality and performance have been demonstrated. By combining the real time functionality with the I/O functionality users can begin to expect to achieve controller level performance from HPUX machines. However, appropriate expectations must be set. The HPUX workstations are multi-user systems. Given enough of an external workload, controller applications performance can degrade to unacceptable levels. Conversely, if a given controller application is running at real time priorities and consuming a large portion of the available CPU resources, then other workloads on the system will be degraded severely.

VI. Real World Considerations

Currently, the HP9000 series 500 HPUX 4.0 operating system is the only HPUX

product supporting user callable DIL. The 4.0 release does not support the interrupt scheme discussed in this paper. However, the HP9000 series 500 HPUX 5.0 release will contain support for interrupts. This release is expected to be available to customers later this year. The HP9000 series 200 HPUX 5.0 release, also expected later this year, is planned to provide all of DIL with the exception of interrupts. DIL is expected to be supported on future HPUX products as well.

One major intention of DIL was to hide as many hardware dependencies as possible. However, this is not always possible. Different hardware presents different problems that must be dealt with. In some cases, when the hardware does not support the functionality as intended, differences do show through. Some of the more noteworthy examples are introduced by differences between the ABI and the TI9914 IEEE-488 bus controllers. These two chips have different behavior that cannot be totally hidden from the user level. The ABI chip does not support setting of the serial poll response byte while the TI9914 does. The TI9914 edge triggers SRQ interrupts while the ABI level triggers SRQ interrupts. These examples and others may be cause for some confusion when porting code from one machine to another. These hardware dependencies will be well documented which will allow application programmers to design portable programs that avoid depending on these differences.

VII. Future Investigations

Within HPUX there exists the possibility of implementing shell level I/O commands. What this means is that a series of I/O utilities could be made available to be executed by the user without writing a program. Shell level utilities could be used to help train new instrumentation users as well as providing some help in early breadboarding of I/O solutions. This could be a valuable aid in designing instrumentation systems given the right set of utilities.

There have been suggestions for looking into faster formating capabilities. The scanf and sprintf do not provide the performance that is generally expected. Atoi and atof do; however, they have no output counterparts.

Another futures item is to implement the complete set of meta-messages. The hpib_bus_cmnd function allows the user to send any sequence of ATN commands, yet it is felt that the meta-messages would be provided to ensure that the commands are sent in the proper sequence.

The locking issue will also be resolved in the future. Locking has been a difficult issue since one can view locking in levels. There could be select code locking and there could be device locking. The difficulty begins when one attempts to determine how the two levels should interact. Select code locking also brings up the issue on how restrictive it should be. Should one allow users to lock the select code with the system disk for example.

Finally, RS-232 functionality issues will be addressed. As mentioned before, a subset of RS-232 capabilities are currently available through the terminal driver and terminfo. However, there has been little feedback to indicate that the functionality available is insufficient.

9003. PERSONAL COMPUTER UNIX for the TECHNICAL PROFESSIONAL: the HP INTEGRAL PC

Andrew L. Rood
Robert C. Cline
James Andreas
Hewlett-Packard Portable Computer Division
1000 NE Circle Blvd.
Covallis OR

1. A Transportable Personal Computer

The goal of the Hewlett Packard Integral Personal Computer (IPC) design team was to produce a transportable personal computer which runs UNIX [1] and which has instrument control capabilities. This paper will assume the design goal. It will not discuss why the goal presumes transportable, why personal computer, or why UNIX. Much debate went into the product definition and many things were considered including modular form factors and other operating systems (e.g. MS-DOS [2],BASIC,p-system [3]). The debate eventually generated a product definition which described a transportable, UNIX-based, personal computer.

The definitions accepted for transportable and for personal computer greatly affected the design of the operating system. The traditional UNIX implementation has some very strong drawbacks for a transportable personal computer product. In particular the design responds to four major aspects of UNIX which make it inappropriate for use in a transportable personal computer.

Transportable was defined as light-weight (less than 25 pounds), and fully integrated in an easy-to-carry package. Transportable implies fully integrated since a modular design is difficult to break down and set up because of the many inter-module connections (cables, connectors). A personal computer was defined as a computer for an individual; a computer which is low-cost, easy-to-use, and tailored to the needs of the single user. A personal computer must be easy-to-use because there is no systems administrator to provide day-to-day help with the PC operation. The machine will be used by an individual who will be an expert in his own field but not a computer expert. A PC must also be low-cost because the machine will be a tool for an individual and must justify its cost to that individual alone. Larger machines have many users to share the expense of the machine whereas the PC expense must be paid by a single user.

HP's reputation for powerful instrument control and our PCD commitment to low cost controllers (e.g. the series 80) implied that the Integral should also provide instrument control capabilities. Traditionally UNIX has been weak in the areas of instrument control and real-time. The IPC adopted the HP-UX/RT (HP-UX Real Time) approach to real-time in the UNIX environment.

These design goals drove the IPC OS implementation: easy- to-use, fully-integrated, single-user, low-cost, with instrument control capabilities.

2. The Integral Solution

The HP Integral Personal Computer is tailored to the needs of the personal computer user. The Integral has been scaled to meet the cost-performance needs of the individual technical professional. The Integral PC provides the UNIX operating system; scaled to fit into the personal computer hardware environment.

The desire for a low-cost UNIX implementation, as well as a commitment to transportability led to consideration of a UNIX implementation without any hard-disc requirements. In traditional UNIX implementations the OS is disc-based; booting from disc and swapping or paging from some system disc. To save the cost of the hard disc a UNIX was configured which boots from ROM using an internal RAM-disc. Further, the UNIX command set has been unbundled to fit onto individual floppy discs of approximately 1 megabyte capacity. The IPC also accommodates external hard discs in the more normal UNIX manner. At the time of the IPC design hard disc technology provided compact high-density hard discs, but these were neither cheap nor rugged; placing a hard disc in a low-cost, transportable package was not within current technology limits.

One of the riskiest tasks undertaken in the Integral design effort was that of making UNIX user-friendly. In the past UNIX has been considered terse and unfriendly to the naive user. In order to overcome this image the IPC design took state-of-the-art user interface technology and applied it to UNIX. The technology involves a windowed user interface modeled upon work from Xerox PARC [4]. Prior to the Integral design effort several PCD engineers had visited Adele Goldberg's lab at Xerox PARC and seen the effectiveness of their windowed, iconic user interface. The IPC design adopted their model to a large extent.

The major part of the creative work of the Integral project involved merging windows and a new user-interface into the OS. This interface uses a custom IC, the Graphics Processing Unit (GPU). This also involved enhancing the UNIX tty concept and structures. Further, the IPC adopted a user-friendly, menu-driven command interpreter; PAM, the Personal Applications Manager (concepts already in use with the HP150). Finally, the integration of the machine, which guarantees that every system will have a built in electroluminescent panel facilitated the construction of built-in high-capability graphics drivers.

The IPC is intended as a single-user machine in the sense that one person will be using the machine at a time. However, the concept of 'single-user' is really a command level distinction. A single user logged onto a UNIX machine can do everything that several users could do when simultaneously logged on. A single user has the capability of running as many processes as he wishes (within machine limits). Single-user implies that the IPC OS design is not as concerned with protection and security as is the OS of a multi-user UNIX machine. Protection from other users is not as necessary in a single-user environment. The ultimate personal computer security is developed when the single user takes his software (and possibly his entire machine) and locks it in his safe.

A single-user personal computer cannot presume that there is a systems programmer who maintains the machine. Rather, the single user will have to be his own system manager. This implied that the Integral should require only a minimum of system management help. The system boots and configures itself automatically. The single-user concept is one which has been refined again and again over the

course of the Integral OS development and is a concept which continues to evolve in the lab today. The concept of a 'single-user' machine has many repercussions throughout the design of the OS and of the entire software/hardware environment.

The IPC concerns with real-time performance derived largely from HP's historical and continuing commitment to instrument control. It was clear at the outset that UNIX would have to be enhanced to provide adequate real-time support. Fortunately, there has been and continues to be substantial effort across the 9000 development divisions to generate and refine an HP-UX/RT enhancement set. The IPC implements those portions of the HP-UX/RT definition which directly address known, glaring, UNIX instrument control shortcomings. In particular the definition enhances UNIX in the areas of scheduling, interprocess communication, and file system manipulation.

This paper highlights four major areas in which UNIX has been customized to operate in the hardware and software environment of the small, personal computer. These four areas involve: operating with a single, small, removable mass storage device; booting and executing from ROM; optimizing utilization of the small, built-in, bit-mapped, display device to provide a friendly user interface; and providing real-time-control system enhancements.

3. Intended Applications

The intended customer for the Integral is the individual technical professional. The Integral is intended to provide vertical solutions in many application areas. Of particular interest are instrument control applications. The intended use also involves HP's Computer Aided Work (CAW) solutions. Computer aided work solutions include word processing, spreadsheet, electronic mail, time management, and other productivity tools. Target Integral users are primarily individual technical professionals in areas where vertical applications exist, and where there is a need for the standard set of PC productivity applications. For example, the chemical engineer who wishes to perform word-processing, graphics presentation, and electronic mail as well as his primary engineering tasks is an ideal match to the Integral. Target users also include UNIX sophisticates such as computer science professionals. The multitasking of the UNIX OS provides capability unmatched by single-tasking (MS-DOS) machines. The Integral allows simultaneous operation at some control task and use as a computer aided work station.

The transportability of the Integral also responds to the needs of the IPC target customers. It is intended that the machine be used in environments where transportability is beneficial. Applications of this sort may involve large labs or factories where the machine is transported among many work stations each day. Another typical application of transportability is for the technical professional who wants to take the machine home with him in the evenings; or the professional who wants to transport the machine with him as he makes trips to various different sites.

The IPC is intended as a single-user machine meaning that at any point in time exactly one engineer will be using the machine. However, over the course of a day or week several different engineers may use the machine at different times.

4. ROM OS

Historically PCD has produced machines with ROM based software and operating systems. ROMs provide many benefits. They are lower cost per bit than are RAMs. They are very reliable. In particular, a boot-from-ROM system is generally more reliable and quicker to boot than is a disc based system. Further, the ROMs fit very well with the concept of transportability because they are rugged in traveling and reliable in power-on in many different environments (much more so than discs and disc drives). Thus, a ROM OS met the IPC design goals of low-cost and transportability while also being rugged and reliable.

The Integral provides 256k bytes of system ROM. The Integral PC kernel code resides primarily in this ROM space. Furthermore, certain processes (e.g. the command interpreter) are ROM resident.

The ROMs are not without their drawbacks. The most glaring of these is the cost to develop and upgrade. The development cost exceeds normal software development cost because of the time required to manufacture the ROMS, which in turn lengthens the software development cycle. Further, upgrades traditionally require an entire new ROM package which is more costly to produce than a new soft distribution. The Integral OS design has attempted to alleviate the upgrade problem by configuring our ROMs to be software "patchable".

The ROMs are jump-linked together. Every major kernel function is linked to the rest of the kernel indirectly through a RAM vector table. This allows downloading OS patches which replace certain OS functions by intercepting the RAM vectors and redirecting them to the soft, RAM-based, replacement functions. In this manner the IPC can even go to the extreme of downloading a new scheduler while the system is executing; and changing to that new scheduling algorithm on the fly. This mechanism and the normal kernel table linkages allow drivers to be configured into and out of the system while the system is running.

Although memory for the Integral is far more abundant than it has been for previous generations of machines; memory is nonetheless a critical resource. By placing the OS itself and certain process images in ROM, OS RAM requirements are reduced. To further optimize RAM utilization, a two level RAM management scheme is used. With this RAM management discipline the RAM resides normally in the user process RAM pool. When the OS requires it (e.g. to allocate blocks for the RAM disc; or to provide system memory for a new driver, or for a new window instance) memory is moved from the user RAM pool to a kernel RAM pool. Whenever memory is freed back to the kernel RAM pool (e.g. by removing a file from the RAM disc and freeing RAM disc blocks or by destroying a user window) that memory is coalesced into the kernel pool and any available memory is transferred back to the user pool for use in user processes.

Memory is a critical resource in a small machine like the IPC, however in a single-user environment memory utilization should be predictable. That is, if the IPC user runs out of memory then the single user is the one who has invoked those processes which have consumed the memory. In a multi-user environment a particular user has to compete with the machine's other users for memory and CPU resources. This competition leads to unexpected performance delays and unpredictable memory utilization. The predictable nature of the single-user environment is one of the desirable features of the personal computer.

5. Display and User Interface

The Integral is most innovative in its user-interface. This interface makes the traditionally terse and unfriendly UNIX user-friendly. The novice can learn to use the Integral to perform real work in a very short time, yet can extend his quick-learned knowledge into an understanding of how to use the whole power of UNIX. Jon Brewster in his paper: "A new user-interface for UNIX" will discuss this in more detail.

The HP Integral PC provides an electroluminescent display panel controlled by a custom graphics processing unit (the GPU). This device provides a display system which is small in display size but great in power. The size constraint derives from the size of currently available display technologies (256 x 512 pixels on the Integral). The great power of the display derives from its bit-mapped nature and the powerful GPU IC which controls the display. The Integral PC uses the great power of the display mechanism to mask its relatively small size. The primary means of accomplishing this is through extremely efficient use of the available display area. In particular the Integral PC incorporates window drivers which allow optimal utilization of all available display area.

5.1 Windows

The HP Integral PC display is viewed by the OS as a collection of 'windows' each of which provides a view onto some entity or process. A window is simply a rectangular portion of the display area which is used for one purpose. These windows can be of arbitrary size and can be placed arbitrarily on the display device. This allows the Integral PC and the Integral PC user an environment in which the display can be optimally used by the intelligent manipulation of windows.

The HP Integral PC implementation provides a framework in which windows can be used transparently by window-dumb applications. A 'window-dumb' application is one which does not 'know' about windows. A 'window-dumb' application is one which presumes it is performing IO to a standard terminal. The majority of currently existing UNIX applications are window-dumb. Applications which have been written with windows in mind can make use of the full- featured programmatic interface to the Integral OS window manager. The Integral command interpreter (PAM) is an example of such a 'window-smart' application.

6. BASIC and Real-Time

The Integral provides real-time support in the form of a subset of the HP-UX/RT standard definition. The IPC design has selected those portions of HP-UX/RT which directly address major problems with current Bell System V [5] UNIX implementations. Robert Lenk in another paper will be discussing the entire HP-UX/RT extension set. This paper will discuss only those items implemented on the Integral. Each will be motivated and briefly described. Finally, a brief discussion of real-time performance and the Integral implementation is presented.

The HP Integral PC hosts as one of its major applications a PCD BASIC environment.

This BASIC language environment enables the user to perform certain real-time control operations. To facilitate such operations the Integral PC provides a set of real time control enhancements. These include: reliable signals (Berkeley 4.1 bsd [6] signals), real-time scheduling, high resolution interval timers and time of day, shared memory, process locking, efficient process forking (vfork), and file synchronization (fsync). In order to accommodate real-time scheduling the HP Integral PC provides an environment in which the system scheduler itself can be replaced while the OS is executing!

6.1 Reliable Signals

The trouble with Bell signals is twofold. First, there is a critical section between the kernel dispatch of a signal and the user re-establishing a handler during which processes may be killed inadvertently. Second, signals are not stacked so that multiple occurrences of the same signal can be reduced to a single occurrence. These problems are resolved using the HP-UX/RT interpretation of the Berkeley signal facility. In this signal package signals are masked from recurring until explicitly released, thus covering the critical section hole manifested by Bell signals. The HP-UX/RT signal implementation also provides counting signals so that multiple occurrences of the same signal will cause multiple signal dispatches. No signals are lost.

6.2 Real-Time Scheduling

Normal UNIX scheduling is round-robin. The HP-UX/RT definition specifies a form of non-degrading real-time priority-based scheduling which operates above the normal round-robin scheduling. Real-time scheduling is accessed using the rtprio call (real-time priority). In this scheduling paradigm all real-time processes (i.e processes running at real-time priority which have used the rtprio call) are executed before any round-robin processes. Real- time processes are absolutely ordered by their real-time priority. Round-robin processes are time sliced. Real-time processes are not time-sliced and will run to completion.

The Integral does not perform preemptive scheduling. Kernel calls are not interruptible and cannot be restarted. This implies that in the Integral implementation the maximum kernel latency is unbounded, with long reads and writes on slow devices possibly locking out other processes for long periods of time. The IPC view is that in a single-user environment the user has control of all processes and can restrict such long reads and writes during critical sections of real-time processing. Further, the single-user machine has less computational power than do most multi-user machines and typically performs only one 'hard' task at a time (where 'hard' implies using a lot of CPU and IO time). This is another aspect of our definition of single-user; i.e. having computational power matched to one user where there is a presumption that many users will in general require more computational power than will a single user.

6.3 Interval Timers and Timeofday

The normal UNIX timing mechanism is the alarm call which provides a second resolution alarm clock capability. The time call provides a second resolution wall clock. For certain real-time applications millisecond (or even microsecond) resolution is required. Millisecond wall clock and alarm clock capabilities

are provided on the Integral using the HP-UX/RT getitimer, setitimer, gettimeofday, and settimeofday intrinsics. Gettimeofday returns the most exact time the system can return (millisecond resolution on the Integral). Setitimer allows the user to program alarms at the finest granularity the OS can accommodate (milliseconds on the Integral).

6.4 Shared Memory

UNIX has been weak in the area of interprocess communication. To augment the relatively weak communication mechanisms of signals and environment passing, HP-UX/RT has adopted a form of shared memory. Shared memory involves the shmop (shared memory operations), shmctl (shared memory control), and shmget (shared memory get) operations. Using shared memory two or more HP-UX processes may directly access the same portion of physical memory. This primitive inter-process communication method is a cornerstone upon which efficient interprocess communication protocols may be built.

6.5 Locking, Forking, and File Synchronization

To facilitate real-time performance processes may lock themselves in memory. This is accomplished using the plock (process lock) call. This prevents those processes from being swapped to disc or from being moved about in physical memory. This allows such processes to protect themselves from unwanted performance 'glitches' which may occur when processes are moved by the OS.

The UNIX OS performs a certain amount of buffering of information written to discs. This buffered information is written to the disc at the discretion of the OS. To force file synchronization (i.e force writes to the disc itself) the HP-UX/RT definition includes an fsync (synchronize file) call. Fsync allows the user to selectively force buffers associated with a particular file to be written to disc. An fsync call performs action similar to that of a close-open-lseek sequence. The close forces all information to be written to the disc, and the open and lseek restore the file status in preparation for the next read or write.

One of the most common operations for a UNIX process is the fork-exec. For example, PAM performs a fork then an exec each time the user invokes a file based command. The HP- UX/RT definition specifies an efficient form of fork, called vfork (virtual fork) which is intended specifically for this fork-exec (or fork-exit) operation. Vfork creates a new process just as does fork, but with vfork the new process and its parent both share the same process image (data and text) until the child either exits or execs. Thus the user of vfork must be careful that the child of a vfork not corrupt its parent's data space (which it shares until exec or exit). However, the vfork operation itself is much more efficient than the fork operation because the forking process image is not duplicated.

6.6 Performance

The overhead to perform a system call in the Integral HP-UX environment is on the order of two milliseconds. The maximum kernel call latency is unbounded because the kernel is not preemptable. For an arbitrarily long read or write call to a slow device the overhead may be arbitrarily long. User processes

receive signals to notify them of external events. The signal overhead is on the order of twenty milliseconds if a process context shift is required (i.e. to start the process which will receive the signal). Signal overhead for a real-time process which is currently executing is on the order of one millisecond (similar to a kernel call). The actual kernel interrupt latency is on the order of fifty microseconds for high priority interrupts. Using Integral specific solutions it is possible for a user process to perform its own interrupt servicing and achieve this fifty microsecond interrupt response. Finally, the Integral is considered to be a .5 MIPS (million instructions per second) machine. This MIPS figure has been developed by comparison with the rest of the 9000 family on a standard set of performance benchmarks.

7. The UNIX File System and Personal Computers

The UNIX operating system as distributed by AT&T Laboratories is not entirely well suited for a personal computer environment. This is especially true in regards to the file system and its hardware requirements. The file system is designed with the idea that access to a hard, fixed disc is guaranteed. This is not always economically feasible in a personal computer environment. Also, hard discs tend to be rather fragile and thus are not well suited to the informal personal computer environment. Certainly hard discs are becoming cheaper and more reliable, but for now, alternatives must be considered.

Floppy discs are the media of choice for most personal computers since the disc drives are cheap and the media is fairly resistant to abuse. Their drawback in relation to using them with a UNIX file system is that they are easily removed. In other words, their presence can not be guaranteed. The UNIX file system is not designed to handle removable media gracefully. By removing a floppy disc at an inopportune moment, one could easily lose large amounts of data residing on the disc through the destruction of key file system data structures.

The Integral Personal Computer has gotten around these problems by providing a UNIX that does not require a hard disc and goes to greater lengths to guarantee the integrity of information on removable media. Modifications were made to the file system and to the means by which disc data is handled to accomplish this feat. The interface to the file system as defined by the UNIX standard is maintained. Certain aspects of every day use are streamlined by added utilities and automatically performing functions which heretofore were not performed automatically.

7.1 Removing the Need for a Hard Disc

The main requirement to be met with the design of the Integral computer was transportability. This could not be reasonably achieved with a hard disc based system. Therefore, the first task was to remove the requirement of a hard disc by the UNIX system. This was done by emulating a disc in RAM. All the essential elements of the file system on the hard disc required by a UNIX system were placed on this RAM disc. These were fairly few in number and did not lead to the use of large amounts of memory.

7.1.1 The Root File System What is known as the 'root' file system on a UNIX

system is the main reason why a hard disc is required. The root file system contains much of the information about how a particular UNIX system is configured. Most of this configuration is contained in the directory '/dev'. This directory contains references in the form of file names to all devices accessible by the system. This is the only way in which applications are able to locate devices and access them. The root file system also represents the place where file systems on disc media are integrated into the overall file system of a UNIX system. If viewed as a tree structure, the 'root' file system is the root of the file system tree. Therefore, to add to the file system tree by pointing to files held by a particular disc, one needs to attach a reference to these files to the file system root. This is done by creating a new branch of the tree from the root that points to the files on the disc.

7.1.2 The RAM Disc The root file system on the Integral is contained in the RAM disc. The RAM disc is as fast as any hard disc and is always present with the machine. Each time the Integral is powered up a file system is automatically created in RAM to hold the root file system. All the information required for the root file system is then dynamically created and entered in the RAM disc file system. The boot up process tracks down every device connected to the system and puts the appropriate entry into the previously mentioned /dev directory. Also, each disc device attached to the system is examined for file systems and each file system found is automatically attached to the root file system. The dynamic creation of all this information is actually a benefit to the Integral. In a transportable personal computer environment, the number of devices attached to the computer can vary widely between times that the computer is used. It is indeed easier to configure them dynamically each time a system is powered up versus burdening a user with having to constantly change some more static form of device configuration. Connecting up all file systems on accessible disc devices also aids the user in efficiently bringing up a personal computer and getting it ready for use. A standard UNIX system with its associated hard disc provides none of these services and relies on the user to track the device configuration and what file systems to attach. So, in truth, the use of a RAM disc over a hard disc provides many more benefits than just reducing the cost of the product and making the product more transportable.

There are even more advantages to the RAM disc on the Integral. One unique advantage is that the RAM disc dynamically allocates the RAM it uses and periodically releases back to the system any unused memory. Most RAM disc implementations associated with personal computers take up a fixed portion of the system's memory whether or not the memory is actually put to use for file storage. Another big advantage provided by the RAM disc is its speed. By storing high use utilities in the RAM disc such as editors or compilers, one can achieve a high rate of performance.

7.2 A Secure Floppy Disc Environment

Floppy discs present many problems to an operating system. The ease with which they can be exchanged makes it difficult to ensure that all file information on them is kept up to date. This problem is made worse by the small amount of data that floppy discs hold. This lack of storage space forces users to swap them frequently. Obviously, since a floppy can be removed from its drive at any moment, the information on them can not be absolutely guaranteed. But, a system must try to do the best that it can.

7.2.1 Synchronizing the File System

7.2.1.1 The UNIX Approach A UNIX file system has a set of disc cache buffers. These buffers are used to hold disc information in system memory in hopes of limiting the number of disc reads and writes needed in order to improve overall file system performance. All reads and writes performed on discs are done in blocks. Each block represents some number of bytes of disc information. The actual number of bytes is system dependent. The Integral uses 1k bytes per block. When a disc read is performed, the block containing the information is read into a system buffer. Subsequent reads from the same block result in information retrieved from this system buffer instead of directly from disc. Writes operate in a similar fashion. If a write is performed to a block being held in a system buffer, only the system buffer is modified. Periodic flushing of system buffers synchronizes the disc with the system buffers. This is how a UNIX system maintains disc integrity. The main idea is to synchronize the disc with the blocks held in memory often enough to keep things straight. Such a scheme is good when dealing with fixed, hard discs when it is unlikely that the disc media will suddenly become unavailable to the system. In a floppy environment, the system would have to synchronize the block cache with the discs fairly often to avoid mishap. Given the slow speed at which floppy discs operate, this leads to a sizable amount of system overhead. A truly tough dilemma between file system integrity and system performance is the result.

7.2.1.2 A Better Solution The Integral Personal Computer is shipped with only one built in floppy disc. When UNIX was chosen for its operating system the dilemma described in the previous section of this paper had to be resolved. First, the problem can be reduced to just the area of caching information to be written to the disc. Read caching changes no data and is desirable to keep for performance reasons. However, write caching presents many problems. One missed write to disc can be disastrous. The first cut at the solution was simply to eliminate all write caching, and to perform all file system writes directly to disc. This would have worked if writes to disc just concerned user data. Unfortunately, with a UNIX file system, a lot of the information written to disc concerns maintaining the data structures of the file system out on the disc. Given the slow speed at which floppy discs work, this overhead slowed the system down prohibitively. Something had to be done to limit this disc activity without sacrificing too much of the ability to maintain disc integrity.

The simple solution to this synchronization problem is to do no write caching of file information. This does work and provides a very secure environment, but there is one large problem. System performance falls off dramatically. The reason for this is that UNIX spends a lot of time updating file control structures out on disc. Each of these updates is composed of a write of a few bytes to the various blocks of control information out on the disc. With the old UNIX caching system, most of these writes were to cached blocks and file system performance was not effected much. With a system that writes through the caching system direct to disc each time, this becomes expensive. First, the block to contain the information must be read in so that the information can be updated, then the whole block must be written back out. Quickly, a few writes of a few bytes becomes a few reads and writes of a few blocks of bytes. The solution to this problem is to limit the number of times that file control information is written out without putting the file system at too much risk of corruption. This is a delicate problem. On the one hand, file information

caching needs to be eliminated and on the other hand, system performance must be maintained. The solution is to do a hybrid of the two systems. All user data will be immediately posted to disc. All file control information will be posted to disc only when a file is closed. This only puts at risk files that are open to a process. Because of how the UNIX file system works, only open files that grow in size are really at risk. A file that is only having information read or existing information altered can not be damaged. This greatly reduces the risk of file system damage while still using caching. Also, since it is file control information that is being cached, the operating system is in a better position to react to problems since it is the only entity that uses the cached information. The user is also more in control of the situation and can judiciously use file opens and closes to reduce the time that files are at risk. This compromise between a secure system and an efficient system works rather well and while not providing a completely secure file systems, it does go a long way toward that goal.

7.3 Unbundling the HP-UX Commands

To distribute the entire HP-UX command collection on small floppy discs (< 1 megabyte) the commands have been broken into small working sets. This unbundling allows HP to market the command sets individually and to provide the commands in a manner which is usable in a floppy-based environment. Currently HP provides discs which include working sets for: standard applications (e.g. vi, adventure, ed), operating system utilities (e.g. copy_disc, format_disc), UNIX commands (e.g. cat, date, ps, pwd), operating system enhancements (e.g. serial driver), hardware diagnostics, tutor, Datacomm, documentation tools (e.g. nroff, spell), development tools (e.g. make), C development tools (e.g. cc, as, ld), and many others.

7.4 Auto-mounting on the Internal Floppy

In a traditional UNIX environment removable media are introduced to and removed from the system using the mount and unmount commands. Because the Integral does not presume an experienced systems programmer, and because we expect that floppies will be introduced and removed very frequently, the IPC design has gone to some effort to reduce the mount-unmount efforts required of the user. In particular, mounting and unmounting of floppy discs in the internal drive is performed automatically for the user. A background daemon process is run at system boot which performs this auto-mounting triggered by a disc interrupt from the internal hardware. This interrupt occurs whenever a disc is removed or introduced. The hardware interrupt is transformed by the internal disc driver into a signal to the auto-mounting daemon which then mounts the new floppy or attempts to unmount the removed floppy.

Further, the IPC design accommodates a mount-instance capability in the internal disc driver which allows several discs to be simultaneously mounted to the one internal disc drive. Reads/writes to a floppy which is mounted but not present yield driver busy errors which typically cause running applications to wait for the desired disc to be restored to the internal drive.

PAM also cooperates with this auto-mount mechanism by moving to automounted file systems and showing their root directory when new discs are auto-mounted.

External discs are auto-mounted at power on for all disc drives connected when the machine is powered on. External discs are not auto-mounted at other times although the IPC does provide a scan-discs command which scans for and mounts new file systems on external discs. With these tools the end user need never know about the complication of the mount and unmount commands.

8. Conclusion

The Integral Personal Computer design team's goal was to produce a transportable personal computer with instrument control capabilities. This goal was transformed into the more tractable goal of producing a low-cost, single-user, fully integrated, user-friendly controller/workstation. This goal led to certain features of the IPC design including its ROM based OS with discless boot, floppy disc based tool sets, windowed user-interface, and adoption of portions of the HP-UX/RT extension set. The concept of single-user has had great impact on the IPC design. This impact is evident in the cost of the hardware, in the interface to the software and throughout the OS itself. The final product meets the original design goal. The IPC is the first fully-integrated, transportable, UNIX-based personal computer in today's market place.

[1] UNIX is a trademark of AT&T Technologies
[2] MS is a trademark of Microsoft Corporation.
[3] Copyright 1978 by the Regents of the University of California (San Diego).
Copyright 1979 by SofTech Microsystems, Inc.
[4] Goldberg, Adele and David Robson; SMALLTALK-80 THE LANGUAGE AND ITS IMPLEMENTATION, Addison-Wesley Publishing Company; Reading Massachusetes, 1983
[5] System V is a trademark of AT&T Technologies
[6] Copyright 1979,1980,1983 The regents of the University of California

9004. DATA COMMUNICATIONS VIA X.25 ON HP-UX

Robert D. Gardner
Hewlett-Packard, Fort Collins Systems Division
3404 E. Harmony
Fort Collins, Colorado 80525

Introduction.

Computer Networking has been in the limelight for a number of years, but the networking picture is confusing, and the customer's perception of it is still vague. With the multitude of networking products and strategies, this situation can hardly be surprising. Adding to the confusion are the products that claim to be the answer to everybody's problems, no matter what the needs are. HP makes only a modest claim for its HP-UX X.25 connection package: it allows reliable file transfers between HP-UX and other UN*X machines the world over, and allows terminal connections to them as well.

Function and Purpose of an X.25 Network

An X.25 Network operates on the well-known packet-switching principle, in which routing stations forward packets to their destinations. Initial access to such a network is generally quite simple, usually through an ordinary Bell-type modem to a local phone number. When there is need for more throughput, more sophisticated methods of network access may be used. Such a network sprang from the need to transfer large amounts of data over long distances relatively quickly. It soon became clear that this type of large network was not only an efficient method for moving data overseas, but the ONLY efficient method for doing so. Because of Europe's lack of approved auto-dial modems and poor quality dialup lines, X.25 networks quickly became dominant there.

Perhaps the most important distinction between an X.25 network (accessed directly) and a dialup network is the allocation of bandwidth. An X.25 network allocates bandwidth dynamically - only when you need it. The result is that you are basically charged by the packet. A dialup network allocates bandwidth for your line, and never deallocates any of it until your call is terminated. Thus, the amount of data sent through a dialup network does not determine cost: you pay for connect time, whose rate is based principally on distance.

The PAD approach to X.25 Networking

As one would expect, a network such as X.25 is necessarily more complex than a dialup network, and there are significant difficulties in connecting to one. Fortunately, there is a device which simplifies the task greatly.

The Packet Assembler-Disassembler (PAD) is a device which saves your host computer the arduous task of building and dismantling "packets." A packet is the basic unit of information in an X.25 network. It contains commands, data, error-correction information, routing information, and several additional fields which will not concern us. These packets are passed from station to station on their way to their final destinations. If your host computer were to perform the task of creating packets for a file transfer function, many cycles could conceivably be

wasted. The PAD allows you to offload this process, and provides the host computer with a standard interface: a modem-controlled RS-232 type connection. The host computer needs no new interface card, no new drivers, and very little application-level support. Because of these simple advantages, PAD's have become quite widespread; adding one to an existing HP-UX system can increase its communication scope to worldwide proportions.

HP's Implementation of PAD Access

The advantages and simplicity of PAD connections over architecture dependent X.25 hardware makes the decision of X.25 access method easy: We want to support an X.25 connection for HP-UX, which runs on several different types of processors, and it is difficult to produce specialized hardware for all in an economical and timely fashion. The PAD solution also eliminates the need for kernel support (drivers) in HP-UX. The well known UUCP/CU package of programs was used as a basis for the project.

Protocol

The first and foremost problem with UUCP file transfers is the inappropriateness of the usual UUCP protocol (the 'g' protocol) for X.25 applications. The 'g' protocol simply sends a series of 64-byte packets, each with checksums, waits for acknowledgement from the remote side that each was received, and retransmits if necessary. This is an excellent scheme if phone lines are used, since they are slow and generally prone to errors. However, this scheme is quite bad for X.25. First, X.25 has internal error detection and correction, so the small packets are inefficient, and their checksums are redundant. Next, the delays involved with waiting for an acknowledgement every 64 bytes, when combined with the propagation delays inherent in long distance communications, makes this protocol about as slow as when it is used on ordinary phone lines. Last, since you pay by the packet, you incur significant financial penalties for not filling up packets to their limit. The solution to this problem was to employ a new protocol - the 'f' protocol, which was invented at the Mathematical Center in Amsterdam. The 'f' protocol does not create packets at all, but simply sends the whole file to the PAD, using nothing but XON/XOFF to pace the transmission. At the end of the transmission, a checksum is sent for the whole file to assure its safe arrival. This scheme avoids the delays involved with a send-ack-send-ack scheme, and is also efficient with respect to packet-filling.

Configuration and Logging

Another problem is the PAD configuration that is required each time a call is initiated or received. A default configuration is not suitable for the several types of calls that may be initiated (i.e., UUCP file transfer with f-protocol, interactive session, etc.) Therefore, there needs to be a facility serving the outgoing and incoming sides of the conversation that perform appropriate PAD configuration. On the outgoing side, there is not much problem, since the HP DIALIT module which ordinarily performs modem auto-dialing, is easily adapted to perform PAD dialing and configuration. However, the incoming side does not have such a convenient facility available. The incoming side also has the additional responsibility of logging the caller's address for security reasons. The "getty" program, which ordinarily waits for a call to come in on an incoming line, is inadequate for such a specialized set of tasks. Therefore, a special

getty is used, and is called getx25. Getx25 handles configuration and logging upon reception of an X.25 call.

Ease of Use

On top of the additional responsibilities placed on dialit and getx25, another important capability is needed: a way for them to work in conjunction with PAD's other than the HP2334A, and to be user- configurable to do so. A similar situation arises with all the various auto-dial modems available for use with HP-UX. Most of these modem have widely differing command sets, and so must have separate procedures for performing the dialing. In addition, all the possible modem types cannot be anticipated by HP, and so the DIALIT module is provided with source code so the customer can customize a dialing procedure for a new modem. This module is written in C, and so requires a good deal of knowledge on the customer's part. For the PAD configuration however, a simpler solution was called for, one that did not require C programming. This solution involves the use of an extremely simple command language specifically designed to communicate with devices such as PAD's and modems. The language has only very primitive commands like "send", "expect" and "error." A fragment of a typical "script" might look like this:

```
/ this is a comment
    timeout 10
try:
    send  "dial"
    expect  2 "number?"
    error try
ok:
    send "555-1212"
         .
         .
```

This program fragment attempts to dial a number on a hypothetical modem by the sending the command "dial" to it. It then expects the modem to send back the string "number?", at which point the script sends it the phone number. If the "number?" prompt is not received within 2 seconds, then the script will try again until the global timeout (10 seconds) has expired. It is obvious that this "language" is much easier to learn and program than C.

Summary

The most common use for UUCP is as a vehicle for carrying mail from machine to machine. The HP-UX facility "notes" (generally known as "news" outside of HP) is also responsible for a large percentage of UUCP traffic. Actual explicit file transfers are relatively rare. As an interpretation of these observations, the most important contribution of the X.25 connection is not simply movement of data from one place to another, but increased communication among people in faraway places. HP's implementation is also transparent to the mail/notes user, working below the user-interface layer of UUCP.

Acknowledgements

Thanks to Radek Linhart for his work on this project. Thanks to Hal Prince for

his PAD configuration language, "Halgol." Also, to Mark Laubach for his assistance.

Further Information

1. HP2334A Multimux Reference Documentation (HP Part 02334-90001)
2. HP-UX Serial Network Reference (HP Part 97076-90002)
3. HP-UX Concepts and Tutorials, UUCP chapter (HP Part 97089-90004)
4. "Unix Networking via X.25" Paper presented by Radek Linhart
   European Unix Users Group Conference Paris, April 1985

A NEW USER INTERFACE FOR UNIX [1]: THE HP INTEGRAL PC

Jon Brewster

Karen Helt

Brock Krizan

Jay Phillips

Hewlett-Packard Portable Computer Division

1. A NEW USER INTERFACE FOR UNIX

The term "user interface" describes all interactions between a person and a computer. This includes keyboard layout, display formatting, command structures, and disc handling. A friendly, yet powerful user interface is the goal of the user interface designer.

The design of the user interface for the Integral PC was driven by the desire to make the power of the UNIX operating system available to the novice. Some important constraints were; to not alienate UNIX sophisticates or previous HP personal computer users, allow import of standard UNIX software, and to allow the novice to become more sophisticated in using the computer as familiarity grows.

This paper describes the user interface for the Integral PC. A complete overview of the entire Integral PC product, and a detailed discussion of the UNIX version used in the Integral can be found in Andy Roods paper: "Personal Computer UNIX for the Technical Professional".

The user interface for the Integral PC consists of four parts:

1. HP-windows (window manager)
2. PAM (Personal Applications Manager)
3. A set of friendly utilities
4. The inherent user interface of each application

The result is a visually oriented multi-tasking system that allows each program to run as it was originally intended. The novice user can get by very simply, and the

---

[1] UNIX is a trademark of AT&T Technologies

sophisticated user can be very productive.

It turns out that the type of multi-tasking offered by the Integral PC can be mastered by the most novice of users. For example, in figure 3 (part of a later example) a calculator has been brought up to generate a number needed in a document. There is no requirement to exit the document editor before running the calculator. In fact, the user can "shuffle" back and forth between the editor and calculator any number of times. Once this sort of multi-tasking is gotten used to, a single tasking environment is difficult to go back to.

2. THE ROLE OF THE USER MODEL

During the design of the Integral PC user interface, it was discovered that the user interface could benefit from two parallel descriptions. There is the "user model" or user's view, and the "structured" or implementors view. Confusion for the implementor and the user is likely if these two views are not understood. The question usually asked was: "How can a user interface be designed so that the details are understandable to the novice?" The answer for a powerful environment seems to be: "You can't! (nor do you need to.)"

An Integral PC user gets information only on a need to know basis. There are very few things that a novice user should be told. However, as the user gets more sophisticated, more detailed information is made available. This new information can be the "truth" about old features, or it can be new features and capabilities. New features may take the form of extensions, or whole new concepts. In general this concept is called progressive disclosure.

An example of the way progressive disclosure works for the Integral is found in the system softkey menu. Windows are placed automatically into default positions, with default attributes. After a user is told how to run multiple applications at the same time, and a certain comfort level is reached, the system menu is described. One keystroke (the [System] key) brings up the system menu. In this menu are commands to move, stretch, and hide windows. The user is allowed to improve efficiency by getting the most out of his screen space without having been overwhelmed by details early in the learning process.

The structured view manifests itself as an "External Reference Specification". The information density in such a document can be quite high. For instance, all the gory details about cursor behavior can be in one place instead of

progressively disclosed throughout a tutorial. The structured view is also the design specification. Information in the user view is too diffuse for a design document.

Keeping track of the two views during the design of the Integral PC turned out to be very useful. Many times the structuring of the user manuals would impact the actual design of the user interface. This happened mostly with the friendly utilities. The Technical writers were important contributors to the user interface design.

3. THE USER MODEL

The traditional UNIX user interface is very powerful and reasonably self-consistent. However it is also cryptic, and its main features are not presented in a way that a novice can understand. This is mainly due to the amount of memorization required. PAM provides most of the techniques for making things less cryptic. The window manager provides a model for easy multi-tasking. Both PAM and the window manager do their job by making things visible, and by prodigious amounts of defaulting. The Theory was: make 10% of the functionality visible, supply good defaults, and 90% of the users problems can be readily solved.

Besides the system user interface, there are the application programs themselves. Application programs are the reason that a user buys a computer. Everything else is overhead. Anything done to make the use or importing of applications difficult will turn out to be a mistake. Let's run through an example as a novice user.

First we power-up the system. This happens relatively fast since the operating system, PAM, and the window manager are all in ROM. At power-up all internal and external discs are made available for use without the need for any configuration file. Now let's insert a disc with the MemoMaker text editor on it. Note that the system detects this and instructs PAM to open the top folder of the disc. (See figure 1 for the current screen.) We can run MemoMaker by using the mouse, typing its name, or using the softkeys. (See figure 2.) While running MemoMaker, we can shuffle back to PAM, run a calculator, and send a number from the calculator to our editor. (See figure 3.) Upon exiting both applications, the screen automatically returns to figure 1.

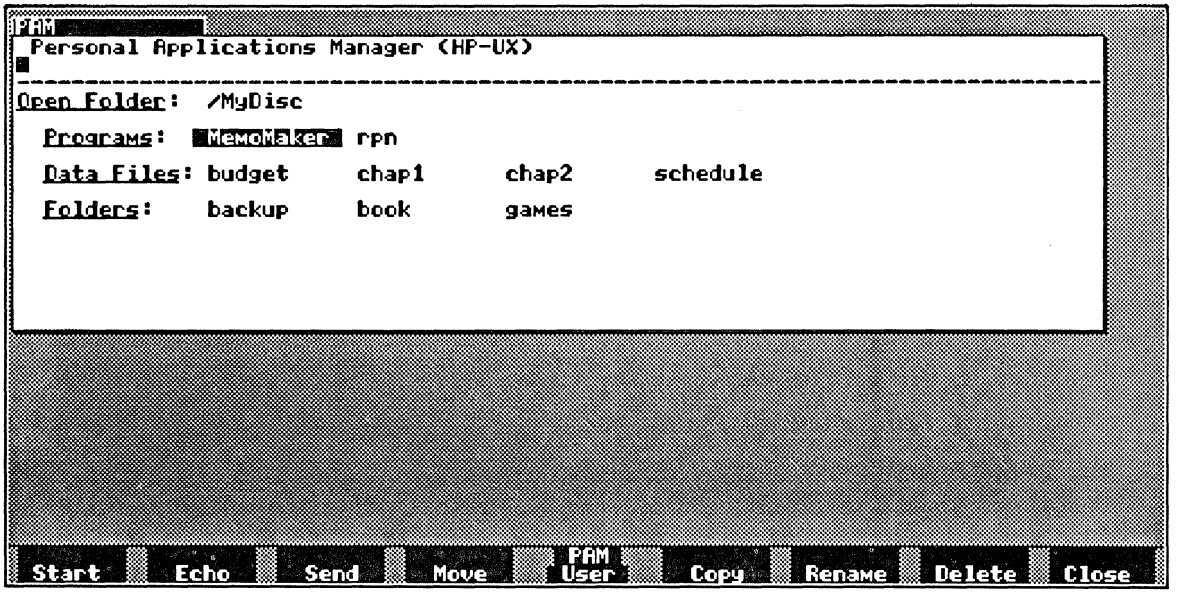


Figure 1

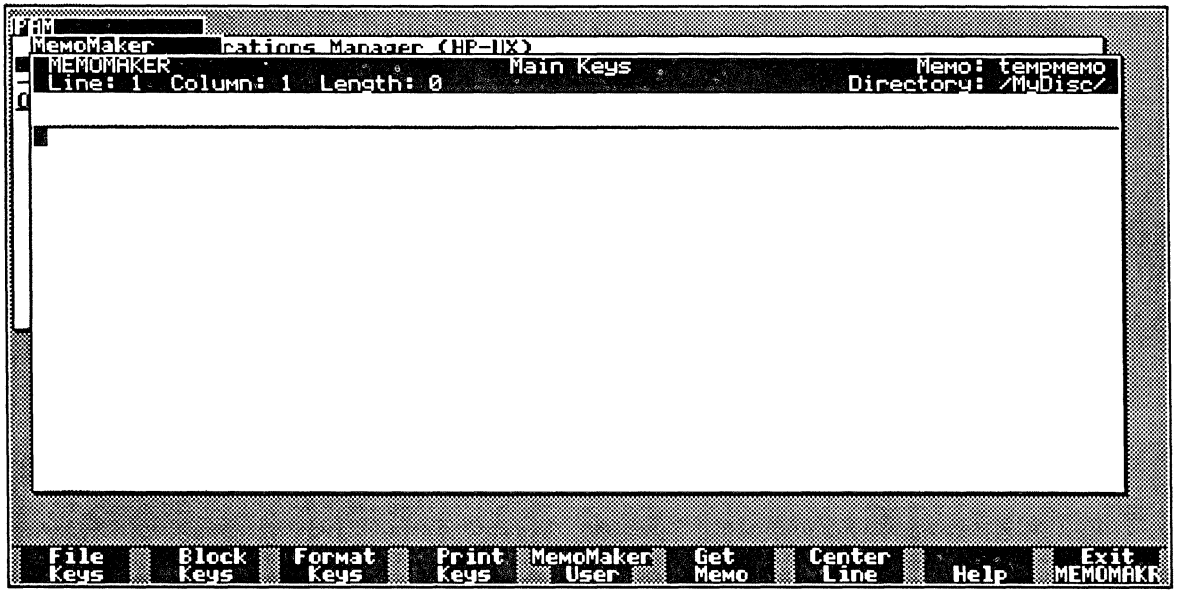


Figure 2

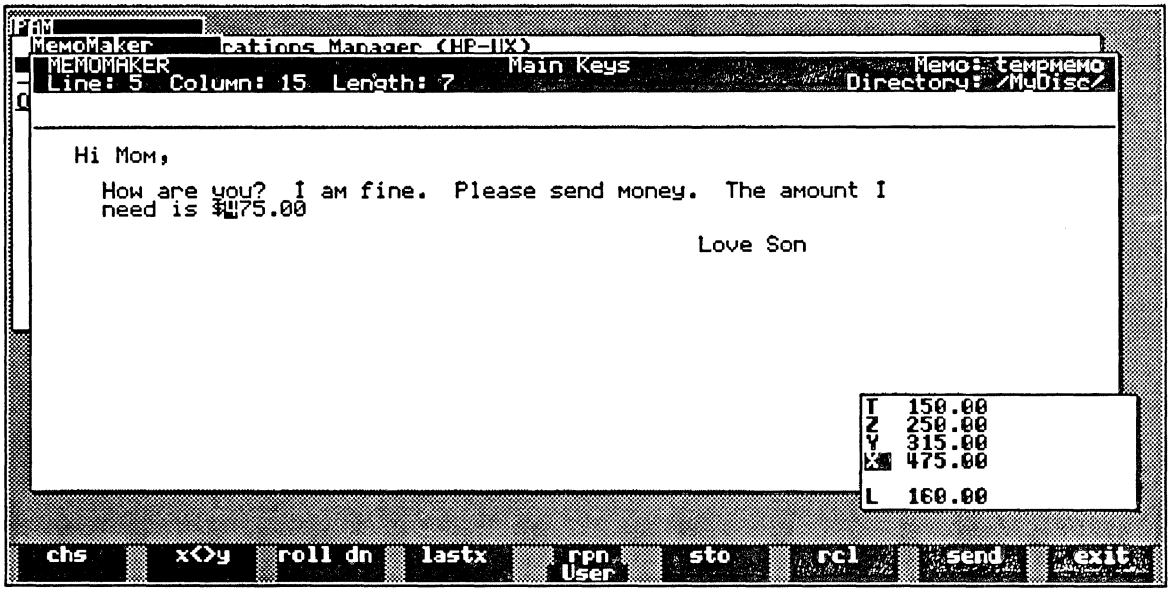


Figure 3

4. THE PERSONAL APPLICATIONS MANAGER

PAM provides facilities for managing files and running programs. Users of PAM have widely varied experience in working with computers in general and UNIX in particular. PAM accommodates this range of users by providing a small basic set of easy to use functions, along with additional functions that are more capable but more complex to use. A novice computer user can run applications and manage data with the basic set of visible functions. Very little knowledge of UNIX is necessary. As the user gains experience PAM's advanced features such as piping and redirection are readily accessible. Experienced UNIX users are not forced to use the basic functions and can use PAM as they would a traditional UNIX command interpreter.

The PAM user interface has three basic parts: the command area, the folder area, and the menu or keylabels. (See figure 1.) With this interface the user can view the contents of a folder, view data files, and enter commands to run applications and manage files. A folder is a collection of programs, data files and other folders. This is also known as a directory.

The command area, which is the upper portion of the PAM display, is where a user issues commands and receives PAM feedback. A command may be a function that is built into PAM (requiring no other program to be executed) or it may be a file in the file system. PAM has built in commands to do the following functions:

- copy, move, rename, and delete a file.
- view the contents of a file
- print the contents of a file.
- redisplay the contents of the open folder in the PAM folder area.
- change the open folder (the folder displayed in the PAM folder area).
- make a new folder.
- type a file name or a line of text in another window.

Commands that are not built into PAM are associated with a file in the file system. This is a standard feature of UNIX shells (command interpreters). With one of the standard UNIX shells the command file must be a program. PAM goes a step beyond this and allows a command to be a program, a data file or a folder. If the command is a program then it is run. If it is a data file then the contents of the file are displayed one page at a time in a separate window. And, if it is a folder, then the folder is made the open folder. In addition to simple command entry (which may be adequate for many applications) PAM supports the following command entry features:

- command parameters.
- specification of the standard input and output of the command.
- specification that the output of a command is to be used as the input of another command.
- start a command and don't have PAM wait for completion of the command.
- start a sequence of commands to be run in order.
- specification of the window in which the command is to be run.

- 20 lines of commands are saved by PAM and are accessible to the user for editing and reentry.

PAM always sets up a window for a non-built in command to use for output. This keeps the PAM display and command output separate and allows PAM to be used while a command is running.

The folder area, which is the lower portion of the PAM display, displays the name of the open folder along with the contents of the open folder. Files of the same type (program, data, folder or device) are grouped together in the display. One of the file names is highlighted - this is the file that is operated on by the function keys and that can be run as a command with a single keystroke or pick of the mouse. The file highlighted can be changed from the keyboard or using a mouse.

When PAM is used to change the contents of a folder or to change to another folder, the folder area display is updated. If the size of the PAM window is changed then PAM will reorganize the folder area display so that it fits within the new bounds of the window. In the event that there is insufficient room to display the contents of the open folder PAM allows the user to display the contents in several parts.

The PAM function keys are used to do one keystroke execution of the most common PAM built in commands. The function keys operate on the file name that is highlighted in the folder area (this file name is the "parameter" for the function key commands). The labels of the function keys and their use change based on the type of the highlighted file (e.g. menu item 1 is used to start a program when a program is highlighted, to view the contents of a file when a data file is highlighted, and to change the open folder and redisplay the folder area when a folder is highlighted). Functions that are not applicable to a particular type of file are not accessible with the function keys when a file of that type is highlighted.

PAM on the Integral has some features that are found in traditional UNIX shells and in PAM on MSDOS machines. But much of its character is based on the packaging of the Integral system and the powerful functionality of its windowing software. Commands to print a file were made to take advantage of the built in printer. Windows are created for applications to use. A mouse can be used to access much of PAM's basic functionality. File manipulation commands are available to take advantage of the built in micro-flexible disc drive.

The command to copy a file, which can be typed as a command in the command area or invoked using a function key, is an example of how a common command is enhanced in the PAM implementation to meet special needs of the Integral user. When a copy command is done with only one file as a parameter the file is first copied to RAM disc and then the user is prompted for the new file name. This allows the user to copy a file from one micro-flexible disc to another even though only one disc can be accessed by the system at a time in the single built in disc drive.

PAM Also has a startup feature that can be used to do system configuration or to run an application. By running an application using this feature the Integral PC can become a "turnkey" system where the user only needs to be familiar with the particular application.

5. FRIENDLY UTILITIES

Even though the user may only want to run applications, there are some ugly concepts that will have to be learned. Some of these are: formatting new discs, back-ups, printer configuration, etc. The friendly utilities represent work done to simplify these jobs. Help options, good defaults, and for the complex utilities form driven input, all help to make these commands as easy to use as possible. Figure 4 shows the copy disc utility.

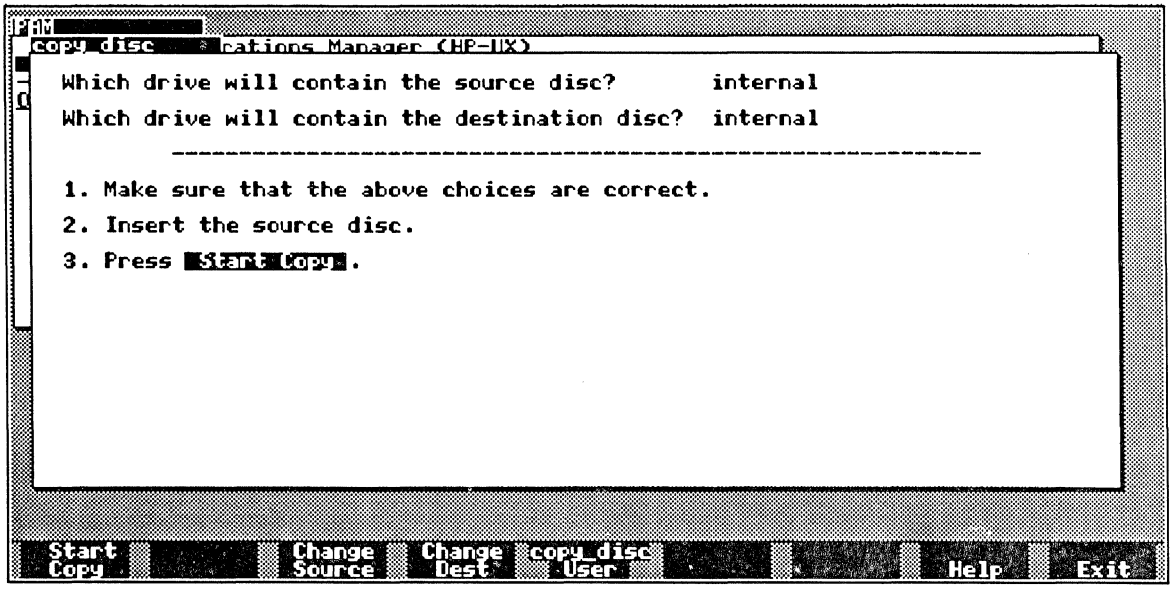


Figure 4

There is also a collection of software that helps make the Integral immediately useful right out of the box. These are the Standard Applications, including: an interactive graphics package, a couple of text editors, an rpn calculator, and games of course.

The Tutor Disc contains an on-line tutorial program. The Tutor is an interactive program that teaches the user how to use PAM, windows, and many other system features. This concept works very well in a multi-tasking multi-window environment.

6. WINDOW MANAGER DETAILS

The first part of the windowing model is the desk and paper analogy. The screen can be considered a desk, with each window a paper on the desk. This model is supported by the fact that windows can overlay each other without modifying each others information. Windows can also be shuffled from front to back, and moved around.

The term "window" is also part of the model. A window supplies a view taken from a large two dimensional object. This object can be scrolled up, down, left, and right to change which part of the object is visible through the

window.

Each window has a name. The name, selected by PAM, is usually the name of the program running or file which is being viewed. The name is contained in a small tab called the window "title" that sticks up from the upper left edge of each window.

Since windows are allowed to cover each other, some rules for uncovering need to be discussed. In general, whenever a window receives new information, either from a running program or from echoing a users input, that window will "show" itself. A window which is showing itself will be in front of the other windows. A window is also shown when it is selected by the mouse, when it comes out from "hiding" (off screen storage), or when it comes to the front via the "shuffle" command.

Windows can also be told to hide. When a window is hidden, the information part of the window is removed from the screen, while the title is placed at the lower left part of the screen. When there is more than one window hidden, there will be a title for each hidden window. (See figure 5.) If output occurs to a window in this state, the window will remain hidden. When the window is next shown (via the selector and the select key) the new information will be there. Note that the running applications are oblivious to all this shuffling, hiding and showing.

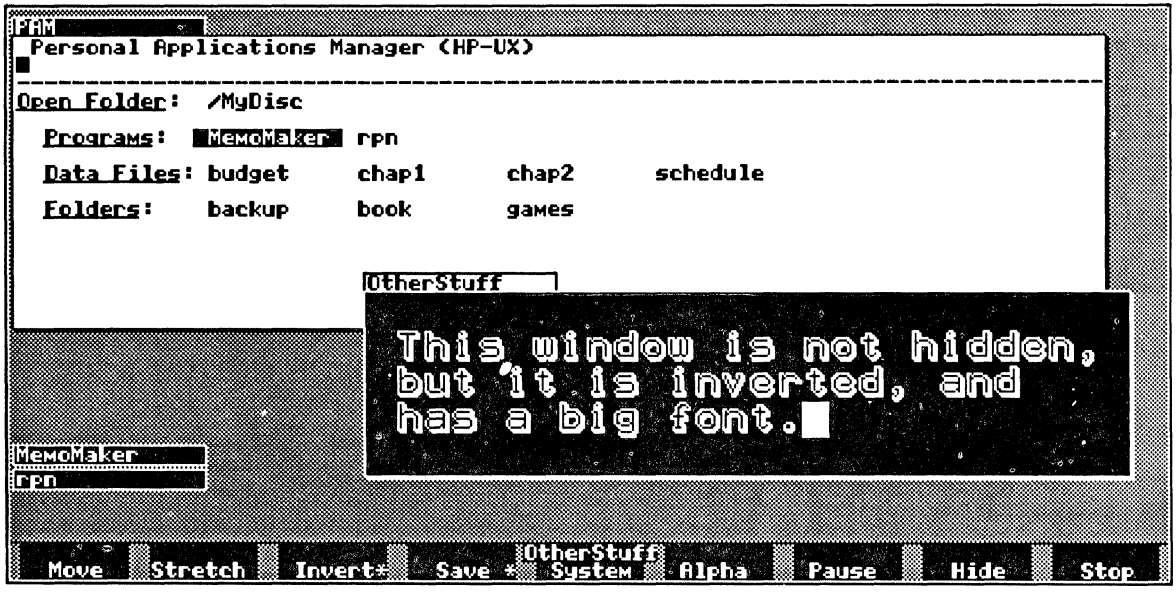


Figure 5

The keyboard is considered to be connected to only one window at a time. When the user types, the characters will only show up in one window. This window is called the active window. The user is in full control of which window is active. A window is made active by the user in the following ways:

1. The mouse can be used to make a window active. Move the selector over any portion of the desired window and press the "Select" button. This is also the mechanism to retrieve a hidden window from the lower left of the screen. Simply select the title.

2. If a window is brought to the top with the "shuffle" command, then it is made the active window.

3. When an application is placed in a new window by PAM, it is made the active window.

Each window has its own cursor. If a window is a terminal emulator (most windows) then the cursor will behave as a normal terminal cursor. If a window is a graphics window, then the cursor is associated with the plotter pen. Cursors are window based images that are not allowed to leave their windows.

The selector is a small image that is free to roam over the entire screen. When the selector is between windows it will appear as a small diamond. When the selector is over the menu it will appear as a small arrow. Each window is allowed its own selector image so that as the selector travels across any portion of the window that image is in effect. The default image is a small arrow. The selector is usually controlled by a mouse, or other pointing device, but can be controlled by the cursor keys (by use of the CTRL key). The selector is used with the "Select" key to select a place of interest on the screen.

The User Interface is designed around a required keyboard and an optional mouse. The following two techniques are used in order to allow the user interface to behave in a consistent manner with and without a mouse:

1. The keyboard can simulate a mouse movement. In particular the cursor control keys can be made to control the selector instead. Simply use the CTRL key with the cursor keys.

2. There are two keys on the keyboard ("Select", and "Menu") that duplicate the functionality of the two buttons on the mouse.

A menu is a list of items equivalent to the keylabels of the Series 80 machines and current HP terminals. These items are listed horizontally at the bottom of the screen in an inverse video area. This area "floats" on top of any windows that extend into the bottom area of the screen. The "Menu" key on the keyboard, and the right button on the mouse, will act as a toggle causing the menu to appear or disappear each time it is pressed.

When a window is made active, the menu will show the items appropriate for that window. The softkeys can be used to access the menu items. However, it is also possible to access the items with the mouse and its "Select" key.

Each window has its own menu called the user menu. This keeps menu items from different programs from interfering with each other. There is also a system menu for window manipulation commands that is shared by all windows. The "User" and "System" keys on the keyboard are used to select the menu type.

The center area of the menu contains the name of the active window and an identifier that tells the user which type of menu is active. (see figure 5.)

7. USER CONTROL OF WINDOWS

Window control can be very simple. PAM creates windows automatically. Unused windows get destroyed automatically whenever a new window is created. The only explicit control a novice user needs is the "shuffle". Shuffle brings the window at the very back to the front, and makes it active. Consecutive shuffles allow a user to review all windows. In particular, even windows which are completely hidden behind other windows can quickly be accessed. The next thing a user learns, is to move the selector with the mouse and use the left mouse button to select a window. With this, a user can make any window which can be seen the active and front most window.

The rest of the user control is performed through the system menu. (See figure 5 for a view of the system keys.)

The [Hide] command removes the active window from the screen without modifying the information it contains. Its banner will be placed in the lower left corner of the screen. To get this window back, move the selector to the banner and press the select key.

The [Invert] command inverts the color sense of the active window. Black on white will turn to white on black, and vice versa. The asterisk will come and go to indicate the current mode.

[Move] is an interactive command that will move the active window to a specified location without changing its size. When "Move" is invoked, the selector will change its image to that of a bracket in the shape of an upper left hand window corner (like an upside down "L"). This corner selector indicates the placement of the upper left hand corner of the window. The user can move the selector anywhere on the screen, and use the "Select" key to finish the command.

The [Pause] command is a toggle that tells the active window to perform no more output until the next pause command is given. The asterisk will come and go to indicate the current mode. When a window is paused, the internal buffers will fill up and then the program will be put to sleep. This is equivalent to CTRL-S and CTRL-Q in a standard UNIX tty driver.

The [Save] command is used to retain old windows that have desirable information. Normally old unused windows get destroyed when the next window gets created. This is to keep the screen from getting cluttered. This command is a

toggle for the active window. An asterisk will come and go to indicate the current mode.

The [Stop] key will destroy the active window if it is not being used (not open). Otherwise, SIGQUIT will be sent to the process group, and the window will be destroyed when/if the final close occurs.

[Stretch] is an interactive command that will change the size of a window while leaving its position unchanged. "Stretch" behaves like "Move" except that the corner selector looks like the lower right hand corner of a window (a backward "L"). The selected position becomes the lower right hand corner of the window, and the upper left hand corner (called the anchor) stays where it is. If the selected position is above or to the left of the anchor, then the window will be placed such that the selected position and the anchor form two corners diagonally opposed to each other.

The [Alpha] and [Graphics] commands bring up window type specific menus. The active window type determines which one of "Alpha, "Graphics", or a future type will be indicated in the system menu. These menus contain commands such as Display Functions for alpha windows, and Pen Up for graphics windows.

Most of the keys on the keyboard are defined by the active window type. There are some keys on the keyboard whose functionality is defined by the window manager.

The [Menu] key is found on the mouse and the keyboard. Pressing this key will toggle the on/off nature of the menu.

The [Print] key will cause a full screen dump to the internal printer.

[Shift][Select] is known as a "shuffle". The farthest window back on the screen will come to the front and become the active window. This is the standard way to switch between windows.

The [Select] key is found on the mouse and the keyboard. When the selector is over a menu item, and "Select" is pressed, that item is invoked. When the selector is at the middle of the menu the menu is toggled between the system items and the user items. When the selector is over any part of a window, then that window is placed on top of any windows which might be covering it, and it is made the active window. When a "Move" or "Stretch" is in effect (as indicated by the shape of the selector), the position on the

screen will be used for placement or size information.

8. PROGRAMMER CONTROL OF WINDOWS

The programmers interface for window control allows for complete control of windows. However, all control is optional. The standard UNIX environment is mostly set up for terminals. The window environment is completely compatible with programs that utilize terminal abilities. In particular, the alpha window type is a pure superset of the normal UNIX tty driver.

The technique for controlling windows was modeled after that which is used for controlling the tty driver. The program makes a call to get a copy of the current state, changes some of the information, and then makes another call to put it back. Some of the things which can be modified are name, location, and size. There are also some control bits which control whether the window is on the screen or hidden, has an inverted background color, is connected to the keyboard (active), etc.

When a program is finished and goes away, the window it was using does not get destroyed. This is because many UNIX programs compute their answers, output, then immediately exit. If the window were destroyed, the information would be lost. Instead, the window is destroyed the next time a window is created (unless the user presses the [Save] key). If a program wants its window to be destroyed when it exits, the program should set the AUTO DESTROY bit. Many times, this is all that is required to make a program "window smart".

Many new programs would like to use the mouse. Asynchronous UNIX events such as alarms, or telephone hang-ups, are handled by signals. The program merely declares that a particular function be called when the event happens. The Integral PC has a new signal called SIGMOUSE. With this signal and a function that reports exactly what happened, a program can easily detect such things as the left mouse button going down, or up, or the window being stretched, or made active.

9. TERMINAL 0 WINDOW TYPE

The main window type in the window system is the Terminal 0 window type (also known as the alpha window). This is the type of window PAM creates for an application when it is started. Since easy porting of applications was an important

goal, the alpha window needed to look just like a terminal to the application.

Applications written for the UNIX operating system expect to talk to a terminal through a serial interface. This serial interface is called the tty interface. The alpha window must thus be a terminal emulator and a tty interface emulator.

The alpha window type emulates a subset of the HP2391 terminal. This subset is known as Term0 or Terminal 0. It has both an HP terminal mode and an ANSI terminal mode. The ANSI escape sequences supported are a subset of the ANSI X3.64-1979 standard, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange". The ANSI mode allows access to additional software which was not written to use an HP terminal. The VT100 terminal is an ANSI terminal, and it is one of the more popular terminals. Thus in ANSI mode, much of the software written for the VT100 could be used. The ANSI mode is not a complete VT100 emulation.

Soft fonts are a useful feature of the alpha window. There can be many different fonts of different sizes in use in different windows at the same time. There is also a font editor shipped with the Integral to allow the user to experiment with fonts.

The alpha window also contains support for fast alpha. Fast alpha is a high performance functional interface that is supported on many HP machines. Fast alpha provides the capability to write strings of characters with or without display enhancements at any position in the window, to place the cursor and turn it on or off, to create, activate and remove fonts, to fill a rectangle with a character with or without display enhancement and to scroll a rectangle in any direction. The performance comes from sending many characters in one call, from not having to look at every character in order to parse escape sequences and from not having to send an escape sequence to position the cursor before writing.

10. HPGL WINDOW TYPE

Importing graphics software was also important for the Integral PC. Since UNIX has very poor graphics standards, an HP standard was selected. It was clear that external HPGL plotters would be supported. So, the graphics window is a reasonably complete HP 7470A plotter emulator. This means that any program that performs graphics via HPGL

command sequences has a reasonable chance of an effortless port.

There is also a set of fast functional entry points for programs that don't care about HPGL.

11. ARCHITECTURE OF THE WINDOW SYSTEM

The overriding constraint on the window manager architecture was the need to emulate already existing UNIX I/O. It would not be acceptable to require a program to be recompiled or order to run it in a window. Implementing the window manager as a normal UNIX program would force much more context switching between programs. It was felt that this would severely impact total system throughput. More expensive memory management hardware is one solution. However, the Integral PC is a personal computer, and keeping the cost down was important. Because of these things, the entire window system has been implemented as a set of UNIX device drivers.

For every window type there is a driver. The standard window manager includes the terminal 0 alpha, and HPGL graphics drivers. There is also a driver called the window control module, which coordinates all screen and keyboard activity. The drivers mentioned above are all independent of the system hardware. The hardware dependent software is contained in the keyboard driver, and display driver. The inter-module connections are shown if figure 6.

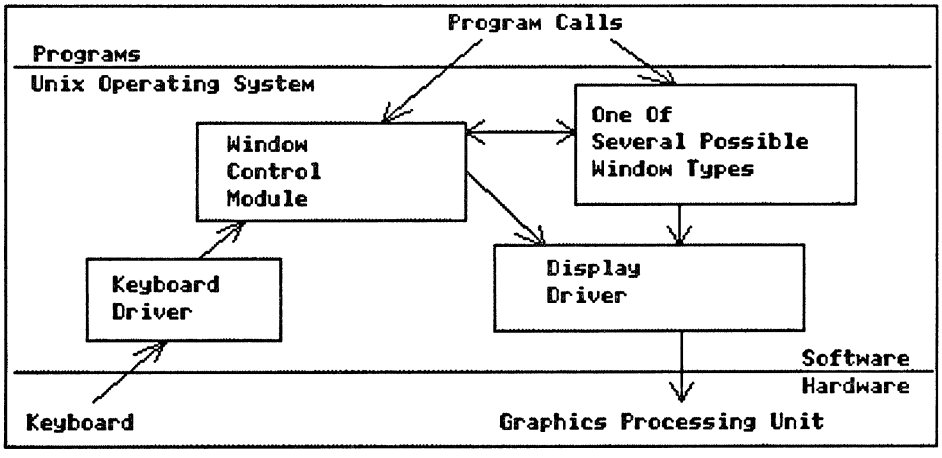


Figure 6

The keyboard driver receives control from the interrupt handler. It processes its inputs into four types:

1. key strokes from keyboards
2. relative motion from devices such as mice
3. absolute coordinates from devices such as tablets
4. ASCII codes from devices such as barcode readers

Keystrokes are mapped through a ram based table into a standard set of 16 bit codes. This table can be modified to customize key layout. This is useful to support Dvorak, and foreign language users.

Once the keyboard driver converts its inputs into a hardware independent form, it calls the window control module for further processing. The window control module processes some of the information (such as shuffle and mouse movement) and passes the rest on to the active window. However, the keyboard driver can also buffer the keystrokes and make them available for direct reading. This is useful for a program that wishes to capture all keystrokes entering the system no matter which window or program would normally receive them. Key strokes can also be written to the keyboard driver. These key strokes will be treated exactly like normal user input.

The display driver has four main jobs:

1. character placement and font manipulation
2. line drawing
3. raster operations
4. identification

At system power up the identification entry points are used to discover such things as screen size and default fonts size. Most of the software in this driver is associated with fonts and rasters. The design of the screen driver had to address many of the classical disc driver problems, such as thrashing, and fragmentation. The main problem is that fonts and rasters are big, and the hardware resources are limited due to cost. Data space for these objects is allocated dynamically from the main system RAM. In order to use these objects, they must be placed into a special area of RAM. This RAM is controlled by the screen hardware. This screen RAM is not large enough for all fonts or rasters

that might exist in the system at any one time. Therefore, the allocated system RAM takes on the role of a backing store.

The keyboard driver only calls the window control module as described above. All modules call the display driver. The display driver never calls other drivers. The window control module and the window type drivers have a more complicated relationship. Each window type driver must have 12 entry points for the window control module to call. Also each window type has at its disposal 10 entry points into the window control module. These entry points are designed to hide the nature of each window type from the window control module, and to centralize screen update and user interface strategies.

The original design, which was worked out in 1982, called for each window type to be a pure abstract data type. The term "window type" dates back to those days. During implementation it was found that some streamlining had to occur. This is mainly in the form of a data structure that both the window control module and the window type code use to find out the current state of things (window location, background color, etc.).

New window types may be added to the window manager. The writer of a new window type must understand the use and maintenance of the shared data structure, the 22 cross module entry points and the abilities of the display driver.

12. SUMMARY

Making UNIX available to the novice user required a great deal of work. Fully half of the Integral PC software team worked exclusively on user interface. The final product is a bonafide UNIX machine that is easy to use.

9006. USING INTERPROCESS COMMUNICATION TO IMPLEMENT DATA BASE CONCURRENCY UNDER HP-UX.

Tim Snider
Statistical Software Group,
200 James St. S.,
Hamilton, Ont., Canada

1. History of SSG

SSG converts statistical and research oriented software to run on HP computers. Originally this was restricted to the HP3000 with all of its code and data space limitations. Although these were overcome for many very large packages, there were some programs which could not be converted until the advent of the HP9000 with 32-bit addresses and virtual memory. One of these packages was the SIR data base management system, a research oriented data base system produced by SIR, Inc.

2. SIR

SIR/DBMS was originally a batch oriented data base system, with a comprehensive data dictionary and very powerful retrieval and reporting languages. More recently, it has been developed into a flexible, multi-purpose data management system with interactive data entry and enquiry in SIR/FORMS, an enhanced version of IBM's SQL language, SIR/SQL+, a soon to be released presentation graphics system, SIR/GRAPH, and multi-user data entry and update via the SIR/MASTER system. All of these have been converted by SSG to run on the HP9000 under HP-UX. It is the last of these that this paper will focus on.

3. Considerations of Portability and Conversion

Most of the software packages that are converted by SSG are written in Fortran. For all of its shortcomings as a programming language, it is still one of the most portable languages available.

In writing portable software it is important to reduce the machine dependancy to the lowest possible level and rigidly define an interface at that level which can be supported on any target machine. If this standard interface is at too low a level, inefficiency may be introduced since some feature of a certain machine may be bypassed. If the interface is at too high a level, it is more difficult to define and may introduce too much variation in the operation of the software on different machines.

All SIR software is written using a macro preprocessor that generates a very portable subset of Fortran-66. The lowest level macros must be defined for each machine that SIR is to be implemented on. For example, all primitive data types are handled through machine specific macros. Thus it is possible to simulate any data type on any machine. In addition to these macros, there are rigidly defined machine, operating system, and file interfaces. These are implemented for each target machine in about 100 procedures and functions, usually written in the assembly language for the machine.

4. Data Base Concurrency

In order to allow concurrent users to update and enter data, it is necessary to coordinate access to data and index files. Modifications to information in data files must be coordinated with other users who are accessing the same data. Modifications to index structures, which occur whenever data is entered or deleted, may have far-reaching effects and must be coordinated with any other users in the data base. This is usually implemented at the operating system level by the file system. Since all file I/O is ultimately done through some form of system call this is an appropriate place to arbitrate between requests from different processes.

This fact caused several problems for SIR in implementing data base concurrency in a portable manner. First, not all operating systems support the degree of file coordination necessary. HP-UX, and Unix systems in general, fall into this class. Second, those operating systems that do support it have a very wide variation in the manner in which it is supported. Third, SIR was already implemented on about many machines, using a file interface at a much lower level than would be necessary to efficiently support data and index locking.

To solve these problems, SIR, Inc. has developed a data base concurrency manager called SIR/MASTER. This package enforces the different levels of data locking necessary and centralizes control over modifications to the index and data structures. User programs make requests to SIR/MASTER to access data, to apply different levels of locking and to write data. SIR/MASTER manages the locks that are in place at any time, arbitrates between users, and is the only program that physically reads or writes the data and index files. A program such as this can be written in a very portable manner, except for the communication of requests and results between the master process and user processes. Thus the problem of data base concurrency is reduced to a problem of interprocess communication.

5. Interprocess Communication

Interprocess communication relies on two components, a communication channel or medium, and a synchronization mechanism or protocol. Under HP-UX there are several choices for both of these. Sometimes, both components are supported by a single mechanism, as is the case where two processes communicate over a pipe. In this situation, the pipe is the medium and the I/O calls on the pipe enforce their own synchronization. A reading process will automatically sleep on an empty pipe and a writer will sleep on a full pipe. Another paper in these proceedings, Integrating Multiple Programs Under HP-UX, describes a situation where pipes are used as the communication medium, but the default pipe synchronization is not adequate. In that case HP-UX "signals" were used. The synchronization implicit with pipes was not suitable for SIR/MASTER either. In this case it was not possible to use signals to correct the problem. Signals may only be sent between related processes. Since many user processes must communicate with SIR/MASTER it is not possible for them to be related.

This is an example of the problems that arise in software conversion. A program written specifically for HP-UX could easily be designed to use pipes with their implicit synchronization. However, in porting a system with certain requirements built into its interface, it is necessary to use something more flexible. For

SIR/MASTER, shared memory was chosen for the communication medium, and semaphores are used to synchronize communications.

6. Shared Memory

Shared memory allows many processes to map a portion of their virtual address space into the same physical memory locations. Data stored in this shared area by one process is available to many other processes.

On the HP9000-500 under HP-UX shared memory is implemented through the extended memory system. When a process uses extended memory an unnamed temporary file is created on disk. References to extended memory access this file through a page table with pages swapped in and out of physical memory as necessary. It is also possible for a process to specify that a permanent named file be used for extended memory. A set of processes that specify the same permanent file name will share the same extended memory. Thus there is no further overhead in using shared memory than for any use of extended memory. An additional feature is that shared memory may be locked into physical memory. This guarantees that it will never migrate to disk. However, memory locked in this manner cannot even be relocated by the HP-UX memory manager, so there may be some negative impact on overall system performance.

7. Semaphores

Semaphores are a synchronization mechanism supporting atomic "test and set" operations. They were originally proposed as a method to support cooperating processes by Dijkstra and are discussed in detail by Brinch Hansen.

The operations currently supported by HP-UX (release 4.02 at the time of writing) are creation and access of individual semaphores, and initialization, increment and decrement of a semaphore's value. Semaphores are created and accessed via a system wide key. Any process using the same key will access the same semaphore. The semaphore invariant as implemented in HP-UX is that a semaphore's value cannot be decremented below zero. Thus a typical usage of a semaphore to guarantee individual access to some resource is as follows. A semaphore is created and initialized to 1 by some master process. This process may now terminate, leaving the semaphore behind. User processes wishing to gain access to the resource may now access the semaphore. Each process attempts to perform a decrement or "down" operation on the semaphore. This operation will test the semaphore to determine whether it may be decremented. The test and decrement are "atomic", that is if the test succeeds the decrement is guaranteed to take place before the process may be interrupted. Thus it is not possible for two processes to accidentally decrement a semaphore at the same time. Any process failing to decrement the semaphore may return a failure indication or sleep until the semaphore is incremented. The one process that succeeded in "downing" the semaphore takes control of the resource. Upon releasing the resource, the process increments or "ups" the semaphore, allowing another process to successfully complete its "down" operation and take control of the resource.

The implementation of semaphores under HP-UX seems to follow very closely to the description given by Brinch Hansen. A semaphore is a data structure managed by HP-UX, having an integer value and a queue of pending operations. When a process attempts a "down" operation, the value of the semaphore is tested. If

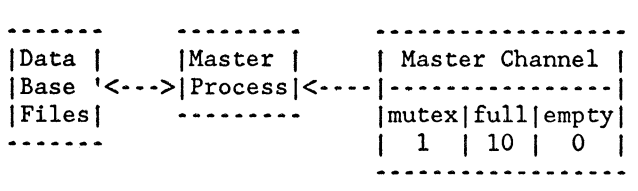
it is greater then zero, it is decremented and the operation succeeds. If it is equal to zero, the process is added to the queue of pending operations and put to sleep. When a process attempts an "up" operation, the value of the semaphore is incremented and the operation succeeds. In addition, if the queue of pending operations is not empty, the next process is removed from the queue, awakened, allowed to decrement the semaphore and continue. The system overhead involved in using semaphores is minimal.

8. Virtual Channels

The interprocess communication interface for SIR/MASTER is defined in terms of "virtual channels". These are one-way communication channels between processes. Processes use a set of routines to open a channel for reading or writing and to read and write a channel. Under HP-UX, these channels have been implemented as circular buffers, using shared memory and semaphores. The buffer is stored in shared memory so that all processes may access it. A mutual exclusion semaphore is used to prevent collisions on updating buffer pointers, etc. Since the semaphore invariant ensures that its value will not go below zero, and it is guaranteed that no two processes will update a semaphore's value at the same time, a semaphore initialized to some positive value may be used to count events. For example a semaphore initialized to 10 will allow at most 10 "down" operations to be performed before an "up" operation. A semaphore such as this which is "downed" each time a message is to be put into the buffer, and "upped" when a message is removed causes writers to be put to sleep when the buffer is full, until a message is removed by a reader. A similar semaphore used in the opposite direction causes readers to sleep when the buffer is empty.

9. Use of SIR/MASTER

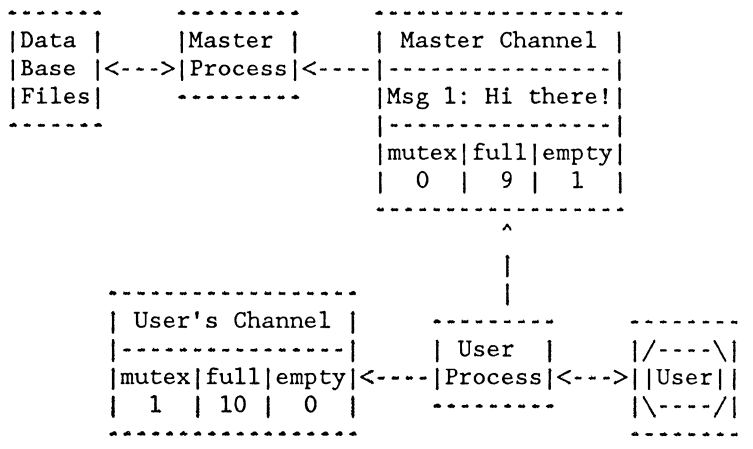
An installation wishing to allow concurrent access to a SIR data base must ensure that a copy of SIR/MASTER is running at all times that the data base is to be accessed. This master process creates the virtual channel that will be used for requests from user processes. The master process will sleep when there is no data base activity since its message buffer will be empty.



A process wishing to use the data base must access the same virtual channel to send requests to the master. It also creates its own virtual channel to receive results back from the master. The user process now sends the name of its virtual channel to the master. The sequence of events is as follows.

- the user process attempts a "down" on the masters "full" semaphore. This succeeds since its value is 10
- the user process gains exclusive access to the masters Virtual channel by "downing" the "mutex" semaphore

- the user process puts its message in the masters virtual channel buffer and "ups" the "empty" semaphore



- this wakes up the master process which was sleeping on the "empty" semaphore

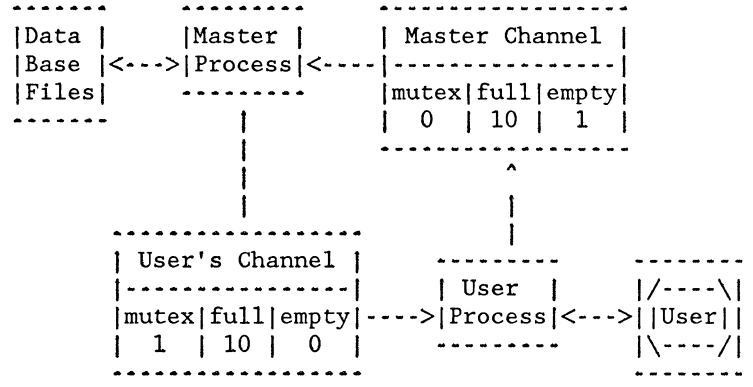
- the master process attempts to gain access to the virtual channel by performing a "down" on the "mutex" semaphore

- this fails because the "mutex" semaphore value is 0, the master goes to sleep again

- the user process having finished with the channel "ups" the "mutex" semaphore, waking up the master process

- the master process accesses its channel, reads the message and "ups" the "full" semaphore to indicate that a message has been removed

- the message tells the master the name of the user's virtual channel so the master may now open it



- the master process now sends an acknowledgment to the user process and communication may proceed in both directions.

10. Conclusions

There are many different methods available now under HP-UX for interprocess communications and there will be even more in the next release. HP-UX 5.0 will support the "msg" internals and a more complete set of shared memory and semaphore internals as specified in Bell System V.2. Each of these offers a different degree of flexibility and ease of use. Each is appropriate under a different set of circumstances. As is usual with UNIX systems, the lowest level functions are available to the user, making it possible to mimic an interface designed for a different system. The implementation of virtual channels for SIR/MASTER was very straightforward using shared memory and semaphores and has been in use for several months without any problems.

11. References

1. Per Brinch Hansen, Operating System Principles, Prentice-Hall, Englewood Cliffs, NJ, 1973.

2. E.W.Dijkstra, "Cooperating sequential processes", F.Genuys, ed., Academic Press, New Programming Languages, York, NY, 1968.

3. Bob Bury, "Lightweight Semaphores", HP-UX Technical Exchange, Vol.1, No.3.

9007. ARTIFICIAL INTELLIGENCE ENVIRONMENTS - SPEEDING UP THE SOFTWARE DEVELOPMENT CYCLE

Sharon Bishop
Information Technology Group
Hewlett-Packard
11000 Wolfe Road, Cupertino, Ca. 95014

INTRODUCTION

Today, there are many different opinions on the future direction of the software market, but one of the few trends experts agree on is that software is becoming more sophisticated, powerful and intelligent. Hewlett-Packard and other companies such as Symbolics, TI, Apollo and Xerox are offering systems which provide both the technology for developing intelligent and sophisticated applications, as well as the environment for execution of such applications.

Hewlett-Packard's first entry into the AI market was presented at the International Joint Conference on Artificial Intelligence hosted by UCLA in Los Angeles during August 1985. Hewlett-Packard's contribution to the AI market is a Lisp-based system that provides both a development environment, as well as an execution environment for AI applications on low-cost, conventional hardware (specifically, the HP 9000 Series 200).

HP's symbolic processing environment, like several others in the market today, although designed for AI system development, also has capabilities that make it valuable for non-AI software development. Therefore, the target market for HP's product encompasses a wide range of users including not only AI software vendors, but government, universities and commercial and technical businesses as well.

You may ask yourself, "why would HP, or any other vendor, for that matter, venture into the AI marketplace?". The two most obvious answers are: (1) the potential market growth and (2) potential internal productivity gains. While there are rampant market projections today, and one must be careful to separate the AI-hype from reality, there is generally a consistent view by market analysts that by 1990 the market for AI applications will have projected sales of at least $5 billion. The various segments that make up this projection include: Expert Systems ($2.5 billion); Artificial Vision ($1.2 billion), Natural Language ($1 billion), Robotics ($500 million) and Voice Recognition Systems ($200 million).

Internal productivity gains is an area that holds great appeal for all companies in today's competitive environment. While software metrics are not easily obtained today to provide a measurement of the dimension that AI techniques may contribute to software productivity, the promise is luring many large companies into activity to investigate and/or develop projects internally. The most well known example of this is probably DEC's XCON tool developed to help DEC configure VAX computer systems.

THE PRODUCTIVITY ISSUE

A programmer/analyst who creates, enhances, upgrades, tunes, tests, or corrects a computer-based application system is performing a highly technical task. The

function being supported, the hardware on which the system runs, the algorithms used, and the flow of information are all complex. Unfortunately, many programmers are still using methods today that are manual -- paging through listings, scrolling through screens and hunting and pecking through the application forest.

Today the productivity problem has reached crisis proportions. The need for software and software engineers is growing exponentially, but productivity is only rising at a rate of about five percent a year. The last significant improvement occurred in the 1950's with the invention of compilers. Now with the mushrooming demand of the private sector and the government, especially the Department of Defense and its Strategic Defense Initiative, there is an even greater impetus to solve the productivity problem. Tools and techniques first developed to support research in artificial intelligence and interactive graphics are increasingly becoming a reality in commercial implementations of software development environments.

Imagine yourself, for example, designing software for a large oil company. You need a system to monitor and control the increasingly complex and frequently changing environment. In an oil refinery, which breaks down crude petroleum into the refined gasoline products that make internal combustion engines run, every step in the process must have a certain yield. You may, for example, have a situation where the refinery control room system operator receives a dozen signals at once that a dozen yields are below par.

Or, imagine that you have been given the task to design a system that will support a nuclear engineer in crisis management. Human response to a crisis in a nuclear reactor requires analysis of the state of the reactor based on a hypothesized failure event, so your system would have to be able to contain a set of hypothesized accidents and a set of event-oriented rules for diagnosing accidents. In the event of a crisis, your system would analyze its own state and conclude what caused the accident and explain it's reasoning to the nuclear engineer.

Numerous other examples abound, from a system to handle I/C design augmentation based on a variety of vaguely stated design rules, to a system that can provide for strategic management of technology planning by coordinating and analyzing complex sets of relationships between technology and products. These examples are presented to illustrate that applications of this nature are increasingly forcing the commercial deployment of a revolution in the way software engineers will have to do business. A common thread runs through the example applications. They are, of course, all large, complex programs whose implementations require significant resources. They all share the similarity that it is extremely difficult to give complete specifications because of the complexity, or continually changing requirements.

CONVENTIONAL ENVIRONMENTS

Today, a typical C, PASCAL, FORTRAN or COBOL programmer laboriously captures the design requirements, using mostly manual tools, entering information into text files, circulating paper and doing very little prototyping before design freeze. Documentation is done by hand or through a word processor, coding is done in a compiled-only language, with explicit edit/compile/link phases, usually using a batch or single-window terminal timesharing system. Most testing and

integration is scheduled late in the project, leaving little room for change, and neccesitating enormous expense to redesign when serious implementation differences exist between the designer and the end-user. Powerful tools today include Programmers Workbench (PWB) UNIX* and the recent Apollo DSEE, with extensive use of Make and SCCS. Recently, object oriented preprocessors and interpreters for PASCAL and C are speeding up the process.

The typical ADA or MODULA-2 programmer is mostly concerned with the role of strict strong-typing interface specifications and module boundaries. Tools interact via a central database serving as system dictionary for definition modules and project database for all code, design and tools. The environments provide the important ability to relate design and specification to resulting code. The database stores different collections of objects, such as alternative configurations, marked with release-date, or as derived from some specification. There is some exploitation of language based editors and "executable" specification languages. Powerful environments such as APSE, ALS and CEDAR have been developed.

AI VS. CONVENTIONAL ENVIRONMENTS

Recently, the terms "evolutionary programming", "rapid prototyping", or "exploratory programming" are used to refer to the kind of systems and lifecycles most associated with SmallTalk, LISP and AI programming. These deal with the interactive and incremental refinement of incomplete systems. Coding, debugging, testing, and maintenance is a single process. Programs evolve via enrichment. Tools include interpreters, debuggers, and language-oriented editors. So called "knowledge engineering environments" aid in developing expert systems. The system captures knowledge in a multi-linked database, which the user can then browse or navigate exploring views and perspectives.

In contrast, conventional programming techniques are ill-suited to handling uncertain or changing specifications. Virtually all modern programming methodology, such as structured design, is targeted to ensure that the implementation follows a fixed specification in a controlled fashion, rather than wandering off in an unpredictable direction. In a well-executed conventional implementation project, a great deal of internal rigidity is built into the system, ensuring its orderly development.

AI "FALLOUT"

And, of course, all of these environments take for granted many capabilities that are essentially the "fallout" from AI development that took place in the 1960s through the 1970s. For example, in the 1960s, the first computer time-sharing systems were developed in AI laboratories to address a faster means for testing and debugging very complex programs. Word processing was developed in the AI lab in the 1960s and experts systems started in 1965. Bit map displays, the now- popular mouse controls, and the whole structure of display windows were also originated during the 1960s as well as object-based programming and a large number of productivity tools.

Commercial use of these tools originally was discouraged by the fact that they require large computing and storage capabilities, but in the 1970s improved computer performance at lower cost made it practical to start using these tools in conventional programming environments. For example, there has been a dramatic

improvement in recent workstation and interface technology, with great cost/performance benefits for group work. Powerful networked workstations with high-resolution graphics are becoming widespread. Today machines offered by vendors such as Symbolics, TI or Xerox LISP machines, DEC VAX's, Apollo, Sun, Textronix 4040 and Hewlett-Packard HP 9000/237, support AI and LISP at modest cost. These machines provide adequate hosts for radically improved programming environments.

AI DEVELOPMENT ENVIRONMENTS

Object-oriented programming is very much on the rise within HP, as well as within the entire industry. It brings to the programmer a productive and powerful paradigm for software development with languages that address concepts such as code-reusability, data abstraction, encapsulation and generic operations. Three years ago, HP began work in its applied research and development laboratories to investigate marketable AI environments. Today that effort has evolved into an umbrella of technology that is being actively used by well over one hundred people at various HP divisions, as well as by students and faulty at major universities around the country.

HP has stepped into the software development fray by offering on its dedicated Series 200, 68xxx based workstation, a cost-effective way to run larger, more complex applications. The product implementation consists of an integrated LISP + OBJECTS + AI programming + conventional languages programming environment. It supports a flexible window management system and input devices of a standard keyboard, tablet, touchscreen, mouse and touchtone telephone (or users may define their own interface). When used as a development machine, all programming tasks are supported for editing, debugging, testing, version management, distribution and documentation. As a LISP programming environment, it provides an editor mode customized to LISP syntax and indentation. This editor is based on the EMACS editor originally developed at MIT. It is a customizable, extensible, self- documenting screen-oriented, display editor. Users can customize, or mold the system in subtle ways to fit their personal style of editing. Users can also extend the system by adding new editing commands, or changing old ones to fit his/her particular editing needs, while he/she is editing. There is a full library of functions at hand for the creation of new editing functions. Interactive self-documentation facilities support the user in effectively using the generous supply of features. Users can edit in two dimensions on the screen, so the page on the screen appears as the page in a book, with the ability to scroll forward or backward at will through the book. As the user edits the page, the screen is updated automatically to reflect the change. Many screens may be visible and active simultaneously on a single physical display, or multiple screens may be active on multiple physical displays.

Another component of the environment is a large library of tools known as browsers. A browser is simply a tool for the viewing/searching/manipulation of a particular set of items. For example, you may use a browser to peruse documentation, files, source code or applications. You might want to conduct automated searches of documentation, or browse and manipulate the contents of a stack. Browsers can provide a simple, yet powerful, intuitive interface that is useful for handling a wide range of problems. Users are provided a large library of browser construction tools and functions to create their own browsers for their particular applications.

As a conventional programming environment a similar set of commands for C, Pascal and Fortran are provided. These include editor modes, code browsers and interfaces to compilers which permit browsing from an "error log" to the offending source. It is anticipated that future capabilities allowing three-dimensional or "faceted" views and additional browser interfaces to supporting subsystems will provide a state of the art environment for programmers using their language of choice. Other additions such as interfaces to C and Pascal interpreters and debuggers, as well as support for additional languages are planned.

To support all tasks encountered by the programmer, a number of optional user services have been integrated into the software development environment. These include office applications such as electronic-mail, documentation preparation, slide editor and telephone services. For example, if the user is in the middle of editing a document, he can send a mail message or view a program source, and can at any time, jump back to editing the document without changing context.

Quite naturally, the ultimate goal of this powerful software development environment is to develop new applications to solve large, complex problems that previously have been extremely difficult or unsolvable with conventional methods. At HP, some examples of development activity using this technology include knowledge-based or "expert systems" applications in natural language understanding, VLSI design, manufacturing, computer diagnostics, medical advisors and intelligent instrumentation. While the majority of this development in today's environment at HP is proprietary and intended to address our own need for productivity gains, it is clear that much of this technology will be leveraged into future products.

FUTURES IN SOFTWARE DEVELOPMENT ENVIRONMENTS

In pursuit of the common goal to automate aspects of the software development cycle, several approaches using AI techniques have been investigated in leading AI research and development laboratories. Expert systems for debugging and maintenance and the more ambitious approach of "automatic programming" are underway. Examples include the Psi project at Stanford in the 1970's and the recent Chi project at Kestrel Institute. The Programmers Apprentice Project at MIT is another example of the first steps in the search for automatic programming. It uses the model of a senior and junior programmer to examine how two people cooperate and what the ideal division of labor between them should be so that the more senior professional experiences a net increase in productivity. A more comprehensive next step has been outlined in a research project called the Knowledge Based Software Assistant (KBSA).

At HP, as our AI technology continues to permeate through the company, we expect greater interest and use of these AI tools by software developers. We are planning the next steps needed to produce more powerful aids to the group software develop process. HP assumes that programmers will continue to work in groups and develop code in a variety of languages on a variety of machines. We are working toward a uniform programming environment, supportive of a multi-lingual, distributed software development process. We feel programming environments will evolve much as operating systems have done. For example, todays operating systems manage a multitude of system resources for users, but requires far less knowledge or interaction than before through the use of unobtrusive virtual memory replacing manual overlays.

The environment will take an increasing role in monitoring and controlling design, coding and debugging of software. Rapid prototyping, reusable tools and a stockpile of reusable code components, along with a full integrated environment to support a programming team and manager with office tasks, program management and tracking, integration and customization of the power of the workstation on a network will be offered.

Object-oriented databases and intelligent software "agents" will coordinate program team activities. The uniform programming environment will become more intelligent about the programming process, and programming "assistants" and smart "apprentices" will control the interaction between the programmer and the underlying language system of choice. The smart apprentice will help to track usage and changes to individual functions, files and systems. The system will know about the language and perhaps have a model of the programmer. The "apprentice" may be able to fill in the missing steps, advise on alternative implementations and do other transitions such as derive more efficient code, expand on specifications or customize generic applications.

SUMMARY

To summarize, HP thinks that we need to make a dramatic improvement in productivity, but to do so, we have to build systems that are capable of exploiting knowledge about programmers and projects. We believe that even though the AI marketplace is in its infancy, there is a great deal of market potential for integrated environments that will provide tools for programmers developing large, complex applications that are difficult or unsolvable by conventional programming methods.

The pressure to increase productivity and avoid a shortage of software engineers is a major factor in driving the move towards sophisticated workstation environments using AI tools and techniques. As hardware costs continue to drop and personal workstations increase in power, programmers will expect more capable and easy-to-use systems. Tightly integrated environments with LISP, PROLOG and object-oriented languages will infuse the conventional software development market. Architectures will continue to evolve, providing an efficient implementation of these AI tools, since performance has long been a concern. Software metrics need to be used to capture the value and increase in productivity from use of these environments, since a long learning curve may be associated with them.

HP's earliest step is to provide an integrated environment based on AI methods where powerful programming tools and office tasks can share a common set of basic tools and interfaces. In the future we will look toward effective object-oriented databases so that many new tools can exploit the accumulated knowledge about programs being built. Eventually, with a firm base, we can begin to build truly "intelligent" expert tools to aid in automating the software development cycle.

REFERENCES

D. R. Barstow, H. E. Shrobe, and E. Sandewall, "Interactive Programming Environments", McGraw-Hill (1984)

S. K. Bishop, "AI in the Commercial Marketplace", IUG Conference, Amsterdam, (April 1985)

M. R. Cagan, "What is PRISM?", HP Software Productivity Conference, Cupertino, (April 1985)

M. L. Griss, "The Next PRISM Programming Environment - Applying AI to Software", HP Software Productivity Conference, (April 1985)

K. A. Frenkel, "Toward Automating the Software-Development Cycle", Communications of the ACM, (June 1985)

T. P. Kehler, G. D. Clemenson, "An Application Development System for Expert Systems", Systems and Software, (January 1984)

Arthur D. Little Decision Resources, "Frontiers for Technology Development", Decision Resources, (November 1984)

James Spoeri, "AI Environment Speeds Software Development", Systems and Software, (August 1984)

9008. REAL-TIME FUNCTIONALITY IN HP-UX

Robert M. Lenk
Hewlett-Packard
Fort Collins Systems Division
3404 East Harmony Road
Fort Collins, Colorado 80525

I. Introduction

There are a variety of UNIX systems and imitations in the market, so it is difficult to make general statements about their strengths and weaknesses. The last implementation which is a common ancestor of almost all current ones was known as Research Version Seven (or V7), which was released by Bell Laboratories in 1978. References to "standard" UNIX systems will apply to this version, as its feature set is common to most systems available today.

The definition we will use of a real-time application is one which must reliably interact with or respond to entities outside the computer itself in a time schedule which is driven by those outside entities. This definition is extremely general. In fact it can be applied to virtually any computer application, as any application will eventually exceed the patience of its user if it does not provide results in the user's time schedule, whether that schedule is on the order of microseconds or years. Therein lies the key to the definition: real-time does not define a black-or-white distinction which can be applied to applications or to operating systems, it defines a continuum of requirements.

Near one end of the continuum are applications which need to respond to discrete inputs, such as keystrokes, from humans. The generally accepted response time for such applications is about one tenth of a second. Somewhat more demanding are applications which respond to continuous input, such as movement of a mouse, from humans. The required response time here is in the range of 15 to 33 milliseconds, in order to track these movements at rates of 30 to 60 hertz. Beyond this, applications which must respond to various instruments, machines, or other computers can require response times on the order of single milliseconds and below.

In addition there are differences in the absoluteness of these requirements among applications. When dealing with humans slight failure rates can go unnoticed, while major fluctuations render an application useless. In other applications failures can cause loss of data or imprecision in control of a machine. In the worst cases such failures can be catastrophic.

Standard UNIX systems, unless loaded beyond the capability of the machine, are generally good at interacting with humans via terminals. However, they tend to have problems supporting applications with more stringent real-time requirements. The way to get optimal performance from a specific machine for a specific application is to write a custom operating system. However, this is a very costly approach, which provides very little leverage either for porting the application to other machines or to porting other applications to the machine of interest. The differences between these extremes are analogous to those between use of a standard high-level language and use of assembly language for

a specific task. HP has traditionally provided proprietary operating systems with very good real-time characteristics for given machines, such as RTE for the 1000 family and the BASIC and PASCAL language systems for the desktop computer family. This approach can be seen as a compromise between the extremes; using the programming language analogy it is similar to use of a non-standard high-level language. The approach being taken with HP-UX is to maintain the industry standard interface, but extend it as necessary to support real-time applications. This is a different compromise, analogous to adding extensions to a standard programming language. The extensions made to HP-UX are intended not to immediately cover the entire continuum of real-time needs, but rather to gradually extend its capabilities to a broader range.

A number of specific deficiencies in standard UNIX systems have been addressed in the real-time extensions to HP-UX. Since the basic HP-UX definition has evolved to become a superset of System V, some of the solutions have come as a part of tracking that definition. Other solutions have been chosen from 4.2BSD, with a slight modification in one case to fit in and preserve compatibility with AT&T and earlier HP-UX systems. In cases where no solution was available in another available system, a new one has been carefully defined to fit in with HP-UX. The first two products implementing these extensions are the 5.0 releases of HP-UX for the HP 9000 Series 500 and Series 200 computers, produced by HP's Fort Collins Systems Division (FSD).

II. Time Resolution

One of the key limitations in the standard UNIX feature set is in the resolution of the primitives which deal with time. The standard primitives for scheduling events at a given time have a resolution of one second, which is insufficient even when dealing with humans.

A feature known as **interval timers** was taken from 4.2BSD to address this problem. Each process can schedule either a single interrupt or regularly repeating interrupts with whatever precision the underlying hardware and operating system support. The interval is expressed in units of seconds and microseconds, in order to keep the interface portable despite the system-dependent resolution. In the 5.0 releases, the supported timer resolutions are 20 milliseconds on the Series 200 and 100 milliseconds on the Series 500.

The BSD primitives for reading and setting the system clock in the same units of seconds and microseconds are also included in the HP-UX real-time extensions. The system clock is maintained with resolutions of 20 milliseconds on the Series 200 and 10 milliseconds on the Series 500.

III. Inter-process communication and synchronization

Another limitation is the set of inter-process communication mechanisms available. There are only two mechanisms common to all UNIX systems, pipes and signals. These facilities have various weaknesses in functionality, performance, and reliability.

A pipe is essentially a one way channel through which arbitrary data is passed with the read and write system calls. It provides synchronization by blocking readers when the channel is empty and blocking writers when it is full. Pipes

have a limitation of requiring the using processes to have a common ancestor which sets up the communication channel. Their performance is limited by the overhead of the read and write system calls. Most implementations of pipes use disc files with an in-core system-wide cache. Thus when the system is busy there may be additional overhead from disc accesses.

A signal is essentially a software interrupt delivered to a process. For each of several signals, each process is allowed to install a handler, a function called when the signal is received. The mechanism was designed to allow for the handling of exceptions such as math traps and interruptions such as a user hitting a BREAK key. The definition also allows for processes to send signals to one another, providing asynchronous communication. However the definition includes various race conditions when signals are sent repeatedly, when processes attempt to suspend execution and wait for signals, and when processes attempt to change the handler installed for a signal. These race conditions can cause signals to be missed or processes to be terminated.

HP-UX includes three new interprocess communication facilities from System V. All three share an interface which allows communication and synchronization among arbitrary unrelated processes, yet provides protection from unauthorized access. An elaborate semaphore mechanism allows solutions to both simple and complex synchronization problems. A message passing mechanism allows transfer of data without any disc access, and provides features such as tagging and prioritization of messages which are unavailable with pipes. Most important for real-time needs is a facility to provide shared memory among processes. It allows by far the highest communication bandwidth, since data does not need to be copied to be communicated. On the Series 500 and 200 shared memory is paged by default, but can be locked in core to provide optimal performance.

4.2BSD introduced a new signal mechanism to solve the reliability problems in standard UNIX signals. It is modeled after hardware interrupts, and its main contribution is the ability to mask out signals in order to eliminate race conditions. One major shortcoming of Berkeley's new definition is that it does not allow full emulation of the standard signals used in virtually every other UNIX system, including AT&T systems and existing HP-UX systems. This is because of an orthogonal change to transparently restart system calls which have been interrupted by signals. A few minor modifications to this portion of the 4.2 definition yielded one for HP-UX which can completely emulate both the standard UNIX mechanism and the new BSD mechanism. The HP-UX definition allows the user to choose whether an interrupted call is restarted as in 4.2 or aborted as in the standard mechanism.

Another new feature of the 4.2 signal mechanism is the ability to designate a special stack for handling certain signals. As defined this feature is highly unportable, since it relies on the ability to arbitrarily move a process's stack pointer without affecting the remainder of its environment. In particular it is impossible to reasonably implement on a stack-oriented architecture like the Series 500. The major value of this feature is the ability to handle an exception when a process has run out of stack space. Thus the HP-UX definition replaced this feature with a system call that allows a user to reserve a specified amount of space for handling certain signals. It is left to the system implementation whether the space is reserved as part of the normal stack or as a separate stack.

IV. Process priorities

Another limitation in standard UNIX systems is in the degree of control available over process priorities. The UNIX process priority structure was designed for multi-user timesharing systems. As such, its major goals are fairness to all users and acceptable response to users at terminals. In order to achieve these goals, the system dynamically adjusts process priorities, favoring interactive processes with light CPU usage at the expense of those using the CPU heavily. Users are given some control of priorities with the nice system call, but the values specified are actually only one factor in a formula. As a result, it is difficult or impossible to guarantee that one process have an effective priority greater than another. In addition, processes executing within the kernel often have their priorities increased to favor them over any process executing user code. The priorities used in these situations are based only on the kernel code being executed, not the original priority of the process. Thus low priority processes inside the kernel are favored over high priority ones outside the kernel or in different portions of the kernel.

Since these problems have not been addressed in any of the commonly available UNIX systems, a new solution is introduced with the HP-UX real-time extensions. This solution extends the priority model with a new range of priorities, named real-time priorities, and a new system call, **rtprio**, which allows processes to set their priorities in this range. Priorities in the real-time range are not dynamically adjusted by the operating system, but maintain absolute values as set by the user. Any process with a priority in this range is favored over any user process with a priority in the normal range, regardless of whether either process is executing kernel or user code. Furthermore, the **rtprio** call allows processes to read and modify not only their own priorities, but also those of other processes owned by the same user. Processes with priorities in the normal range continue to behave with the standard UNIX semantics.

V. Filesystem performance

One major factor in the real-time response of some applications is the speed at which they can log data to disc files, or read the data logged by other processes. The mechanism used by the standard UNIX file system to allocate file space does not lend itself to optimal performance in this area. File space is allocated only at the the time a write is performed; new blocks are added to the file as needed from a list of free blocks. In the case of a single application writing to a freshly formatted filesystem, this may produce an optimal file layout. However, as files are created and destroyed, the free list tends to become random and the layout of any given file does the same.

Since its inception, HP-UX on the Series 500 has had a filesystem implementation which is better suited for optimal file layout than the standard UNIX implementation. It organizes files as lists of varying-length extents, rather than as lists of uniform-length blocks. Whenever it extends the length of a file it attempts to make a contiguous addition to the last extent, rather than allocating additional blocks from a free list. A bit map of free space on the disc is maintained for this purpose. Also disc space is partitioned in cylinder groups, with small seek times within any one group. The algorithms attempt to allocate space for a given file from a single cylinder group, placing unrelated files in different groups to minimize contention for the same space. The more recent 4.2BSD

filesystem implementation includes many of the same or similar improvements to the standard UNIX implementation. In particular it also uses cylinder groups and a bit-map for allocating free space. It does not use varying length extents, but does use larger block sizes than standard UNIX to speed up transfers. The 5.0 release of HP-UX on the Series 200 uses essentially the 4.2BSD implementation, since it is an accepted industry standard.

The bit-map allocation algorithms will almost always result in more optimal file layouts than the free list algorithm. Even so, when file space is only allocated as data is written, the layouts can be unnecessarily fragmented. This is because the operating system does not know the eventual size of the file when it must allocate the first portion, and there may be contention for the same space from other files being written simultaneously. Thus a new system call, **prealloc**, is included in the HP-UX real-time extensions. As the name suggests, this call is used to preallocate space for a file before the actual data is written.

VI. Control over swapping

Almost all UNIX systems support more processes than can fit in physical memory at one time by swapping whole processes, segments, and/or pages to backing store. The time required to bring one of these entities back into memory can range from several milliseconds to several seconds, and can thus violate almost any real-time requirement. The effect is often a wide variation in the performance of a given task, where the normal time is acceptable but an occasional swap causes problems. Since a process which is idle is generally the best candidate to be swapped out, one which suspends itself to wait for an external event is especially likely to suffer.

HP-UX has adopted a solution to this problem from System V. The **plock** system call allows its caller to lock its executable code and/or its data in memory, to avoid unexpected swapping and paging. In addition, as alluded to above, shared memory segments can be locked in memory as needed.

VII. User access to capabilities

Some of these real-time features, notably real-time priorities and locking of memory, allow users to significantly degrade the performance seen by others on a multi-user system. The standard approach to capabilities of this nature in UNIX systems is to restrict the capability to the "superuser" or system administrator, who by definition has the ability to impact all other users. Unfortunately the superuser has many very dangerous powers, such as the ability to write or remove any file in the system, and care must be taken by anyone running as the superuser to avoid their accidental misuse. Therefore it is not optimal to require users using these real-time features to run as superuser.

HP-UX includes a new concept called **privileged groups** as a solution to this problem. The system has several potentially dangerous capabilities or privileges which are ordinarily reserved for the superuser. However, the superuser can assign any of these privileges to a groups of users, designating that group a **privileged group**. All processes in a **privileged group** are empowered with the restricted functionality. The Series 200 implementation permits assignment of the real-time priority and memory locking privileges, as well as a third privilege not related to the real-time extensions. The 5.0 release of the Series 500 does

not implement privileged groups, but restricts these rights to the superuser.

VIII. Conclusions

A number of features have been added to HP-UX to extend its suitability in the realm of real-time applications. At the time this paper is being written, the first implementations of these extensions have not been shipped, and performance measurements on them have not been completed. Shipments of these systems will have begun before this paper is presented, and it is expected that measurement results can be presented at that time.

Hewlett-Packard is particularly interested in satisfying the needs of its traditional customer base. For FSD, this base has included many users of single-user BASIC and PASCAL workstation products in the area of measurement automation. These real-time extensions, together with features like the Device I/O Library, should allow many applications of this type to be implemented on HP-UX systems. This will promote greater portability of such applications, and also provide their users access to the many facilities available on an industry-standard operating system.

Within FSD's Research and Development Laboratories, these real-time extensions have been used in developing an interactive graphics library and a window management system. The results of these efforts subjectively show a marked improvement over previous HP-UX systems in the ability to smoothly track input devices under human control. These particular applications will prove very important in the ability of HP-UX to support computer-aided engineering.

9009. FAST ACCESS TO HP-UX(500) FILE_SYSTEM

Itzhack Goldberg
CMS Ltd.
11 Masad St.
60706 Tel Aviv
Israel

MOTIVATION:

One of HP's CUSTOMERS located in ISRAEL, was inquiring about our UNIX machines. Their main concern was the UNIX FILE_SYSTEM I/O performance. The deepest concern was that the WHOLE UNIX IO is BUFFERED, therefore even the use of the new 4.2 FILE SYSTEM (also known as the FAST FILE SYSTEM) couldn't answer that requirment, as it uses a BUFFERED I/O as well. A BUFFERED I/O isn't an adequate solution for an I/O bounded application.

The current solution for an UNBUFFERED I/O on a UNIX SYSTEM is: the RAW_DEVICE ACCESS METHOD. The customer wasn't too enthusia solution. He already has his own DESIGNED FILE SYSTEM on an old application. This experience convinced him to prefer a customized SYSTEM over a locally developed one. The RAW_DEVICE ACCESS METHOD actually forces him to implement once more such a SYSTEM.

The use of a WELL KNOWN and SUPPORTED system is preferable to the use of a SPECIAL APPLICATION ORIENTED SYSTEM for a few reasons:

> It is more beneficial to concentrate on the application, instead of spending resources on something that may be bought for a cheaper price. The time allocated to design, write and maintain that FILE SYSTEM can be used on the profitable target application, let alone the fact that any FILE SYSTEM like any SOFTWARE PRODUCT is being changed and updated, which makes the SPECIAL FILE SYSTEM development process an endless one.
>
> The SPECIAL FILE SYSTEM might need its own hardware which makes that alternative a very costly one (This is the case for instance with the HP9000). It may be even worse when the SPECIAL FILE SYSTEM is not using that totaly dedicated disc, in that case it is very difficult to justify such a solution.

However there can be yet a good reason to keep a SPECIAL FILE SYSTEM and that reason is P E R F O R M A N C E . In cases like this, one is bound to develop his own FILE SYSTEM, otherwise his application won't sell.

For a customer who wants to GET INTO THE UNIX WORLD and has an I/O bounded application there are TWO BIG QUESTIONS to be answered:

1. Could his application withstand the I/O PERFORMANCE of the common UNIX environment ?

2. If the answer to the 1'st one is a negative one then: DOES IT NECESSARILY IMPLY A SPECIALLY DESIGNED FILE SYSTEM ??

This paper assumes a NEGATIVE ANSWER for the first QUESTION. It will try to present a way to enable a user to perform an UNBUFFERED I/O on an ORDINARY UNIX FILE SYSTEM, if his application can LIVE with a few necessary restrictions.

A way to avoid the development of a new FILE SYSTEM is to employ the UNIX FILE SYSTEM attributes for a RAW_ACCESS method. In other words if there is a way to ask the SYSTEM about the ABSOLUTE ADDRESS of EACH FILE on the DISC, then if the application won't try to READ after the END OF FILE or WRITE after the END OF FILE, it is possible to access that FILE in a RAW MODE.

The UNIX machine on which this package was developed was an HP9000(500). Although this package may run transparently to the user, it is specific to this machine. This FILE SYSTEM resembles the BERKELY 4.2 FILE SYSTEM but is quite different from it. This means that any SOLUTION developed for it can only prove the FEASIBILITY of such a method. For those UNIX SYSTEMS which use the REAL BERKELY 4.2, this package needs to be rewritten.

To figure out what is the STRUCTURE of the FILE SYSTEM one has to look into the '/usr/include/sys' directory. The result of this phase may be a PROGRAM that enables the users to describe for any given FILE the ABSOLUTE ADDRESSES of its EXTENTS, as well as their SIZES. Such a program is: DESIND(1); for more explanations refer to the appropriate manual. Here is an OUTPUT EXAMPLE(copied from the manual page for desind(1)) of the command:

desind /bin/sh

```
desind: /bin/sh     D E S C R I P T I O N
----------------------------------------

FILE SIZE in BLOCKS = 39
FILE SIZE in BYTES  = 39556
Media BLOCK_SIZE SIZE in BYTES  = 1024
-------------------------------------

EXT_NO              START_ADDR          EXTENT_SIZE
------              In BLOCKS           In BYTES
------              ----------          -----------

  0                    59882             32768
  1                    55336             7168

FILE SIZE EXTENTS NO. = 2
-------------------------
```

Having a DEBUGGED 'desind'-like PROGRAM is essential before moving to the NEXT PHASE: usage of the information obtained and demonstration which will exemplify the unbuffered I/O superiority to that of the HP-UX buffer oriented I/O method.

In order to make the RAW_UTILITY a user friendly solution, TWO POINTS have to be TAKEN CARE OF:

1. The use of the RAW_UTILITY should not disable the possibility to

PORT the USING program to another UNIX SYSTEM which lacks the RAW_UTILITY.

2. The UNBUFFERED METHOD HAS TO BE A CLEAN METHOD. It shouldn't VIOLATE the UNIX FILE SYSTEM SECURITY at the user level nor at the system one.

Another point, not less important: Bench-mark results should illustrate the SUPERIORITY of the UNBUFFERED RAW ACCESS method to the regular BUFFERED I/O of the UNIX. Without firm proof this package won't be used. A study has to be conducted which compares different SYSTEM CONFIGURATIONS as well as buffer sizes. The results should relate the transfer rate to the system configuration and the different buffer sizes.

In the following sections these POINTS will be COVERED.

PORTABILITY:

The RAW_UTIL PACKAGE contains FIVE BASIC SUB-ROUTINES and the APPLICATION PROGRAMS call them. The 5 SUBROUTINES are: MOPEN(3C), MCLOSE(3C), MWRITE(3C), MREAD(3C), MLSEEK(3C). The program '{M|m}cp' for which the source is included, may serve as an example for the way the PORTABILITY PROBLEM CAN BE SOLVED. It can be COMPILED in mainly two different environments.

If it is compiled on the HP-UX9000(500) you just type the following line:

```
cc -v Mcp.c -lraw -o MCP
```

And the resultant 'MCP' then being changed own to the SUPER USER ownership, then execute: chmod 4555 Mcp (As a SUPER USER OF COURSE). Now one has the {m|m}cp which utilizes the UNBUFFERED RAW_ACCESS METHOD. A valid test of its performance is a file copy of a large file. However if the {M|m}cp has to run on another UNIX machine, that machine just needs to have the system call prealloc(2) in its KERNEL.

Then just type the following:

```
cc -v -DORDINARY Mcp.c -o Mcp
```

And the resultant 'Mcp' will be running on that system as well. The DIRECTIVE 'ORDINARY' mainly tells the PRE-COMPILER to insert for EACH Mcommand its sys call; i.e., for 'mopen' there will be the sys-call 'open(2)' etc.

By the way the {M|m}cp introduces a NEW feature the '-P' which if NOT given will direct the program NOT to OVERWRITE an ALREADY EXISTING FILE. For a more in depth understanding of the 'ORDINARY' directive usage refer to: 'Mcp.h' an include file associated with the {M|m}cp PROGRAM.

Security:

The clarity of the solution involves the following:

> For one: Will those programs which use the UNBUFFERED RAW ACCESS method endanger the FRAGILE SECURITY OF the UNIX FILE SYSTEM ? In other words could a process either unintentionly or on purpose OVER-WRITE an area on the disc which it is not supposed to ?
>
> Secondly: If the {M|m}cp is taken as an example for a USER application; that program has to have the capabilities of the SUPERUSER, which enables the PROGRAM to carry out the UNBUFFERED RAW ACCESS METHOD to the DISC . How does one make sure that ALL UNIX FILE SYSTEM REGULATIONS ARE OBSERVED ? In other words: Could I COPY a FILE that I'm not supposed to ?

In the following sections these questions will be addressed.

As for the DEBUG level the RAW_UTIL had gone through, it's a fact that the UTILITY was developed and tested under a normal HP-UX FILE SYSTEM. Except for the last PHASE of the UTILITY testing, all development and debugging was done on A ONE HP-UX NATIVE SYSTEM DISC. Not even ONCE did it cause any sort of SYSTEM FAILURE. It NEVER OVER-WROTE an unwanted file or by large it never caused the FILE SYSTEM any damage . That SHOWS that the MAIN BUGS were FOUND.

A program which runs under the EFFECTIVE SUPER USER ID CAN READ or WRITE FROM/ON any FILE on the SYSTEM. To avoid that OVER QUALIFICATION of a PROGRAM, one has to USE the system call ACCESS(2). The MOPEN(3C) VERIFIES if the requested FILE SHOULD be MOPENED or not. This way the DESIND(1) makes sure one can't interogate a file which he doesn't have the access permission set to allow it for himself.

In the {M|m}cp there are also some access(2) calls because the command is RUN under the SUPER USER mode. This STRESSES that the SUPER USER who is responsible for the different PROGRAMS RUNNING in the SYSTEM under HIS EFFECTIVE UID, DOES HAVE to CHECK and DOUBLE CHECK the APPLICATIONS which receive HIGHEST ACCESS PERMISSION in the ENTIRE SYSTEM. This concern is true for any application and NOT JUST FOR THOSE WHICH USE the UNBUFFERED_RAW_ACCESS_UTIL.

Another ISSUE to address is: From where does the restriction of _MFILE arise?

This restriction results from the fact that during the MREAD(3C)|MWRITE(3C) operation, the SYSTEM is not aware of the fact that the UNBUFFERED_RAW_ACCESS FILES are being accessed. A situation like this may cause a BIG PROBLEM of FILE SYSTEM INTEGRITY.

Suppose that another user or even the same one is not aware of a BACKGROUND process which tries to READ or WRITE from/to a CERTAIN FILE of his, and by mistake (or intentionally) REMOVES this FILE.

Under the regular UNIX environment the file will be PURGED only after all file descriptors are closed. This way, UNIX ensures the integrity of the FILE SYSTEM and AVOIDS LOSS OF DATA. But in the case of the MREAD/MWRITE the FILE SYSTEM MANAGER is actually bypassed. The FILE would PROBABLY be REMOVED and the PROCESS

will, in the less difficult case, MREAD irrelevant INFORMATION which could belong to a newly created FILE, or under the WORST case our process will OVERWRITE the newly created FILE.

In order to avoid a situation such as this, the MOPEN(3C) also opens the FILE as a REGULAR FILE so the SYSTEM will be notified about the FACT that SOME ONE is DEALING with the FILE and POSTPONE the ACTUAL REMOVE until ALL FILE DESCRIPTORS related to the FILE are CLOSED.

The following TABLES discuss performance benchmarks which were performed:

++

The SETTING of the HP9020 while running the tests:

The machine was in a single user mode.
The SYSTEM WAS loaded with 2.25 MBYTES of MEMORY
Cache buffer number = 120
Read_ahead_level = 4
Page_size = 1024
The COPIED FILE called BIG is a 10Mbytes size.
The tested DISCS were HP7912 and HP7914.

MCP BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
1K	12:19	4.7	49.3	5:14	5.2	49.0
2K	6:21	2.5	25.7	2:55	2.4	27.0
4K	3:29	1.4	14.8	1:42	1.2	14.5
8K	1:56	0.5	8.8	1:05	0.6	8.7
16K	1:12	0.3	6.3	47.7	0.3	5.9
32K	50.8	0.1	3.9	39.0	0.2	4.6
64K	46.3	0.1	3.9	35.3	0.0	3.8
128K	35.0	0.0	3.4	32.4	0.0	3.6
256K	32.1	0.0	3.4	31.5	0.0	3.4

At that CONFIGURATION the REGULAR READ/WRITE application figures WERE:

BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
1K	1:25	1.2	41.4	57.1	1.3	42.5
256K	1:24	0.0	30.6	1:20	0.0	32.1

The most impressive and OUTSTANDING DATA is certainly the fact that the REGULAR WRITE/READ were almost indifferent about their BUFFER SIZE.

One could think that the reason for that was the complete dedication of 120 CACHE BUFFERS to the SINGLE USER TASK. In order to evaluate the performance and compare the MCP and the REGULAR READ/WRITE COPY the configuration was changed so that the BUFFER NUMBER WERE REDUCED TO ONLY FOUR . The following illustrates how the performance was effected:

MCP BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
256K	37.2	0.0	3.9	32.6	0.0	3.3

REGULAR BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
1K	9:20	1.1	45.2	2:32	1.1	44.3
256K	2:40	0.0	32.0	2:36	0.0	30.3

These DATA doesn't represent any REAL MIXTURE but this TIME the OUTSTANDING DATA is the FACT that the UNBUFFERED_RAW_ACCESSED bases MCP was hardly affected in its PERFORMANCE while the ORDINARY WRITE/READ were indeed.

The NEXT STEP is to enlarge the NUMBER OF THE CACHE BUFFER NUMBER to 256 and CHECK how this change had affected the two cases:

MCP BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
256K	31.4	0.0	3.2	32.5	0.0	3.5

REGULAR BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
1K	1:32	1.3	42.6	1:05	1.4	43.6
256K	1:23	0.0	31.8	1:19	0.0	32.4

Once again one can CLEARLY see as EXPECTED that the UNBUFFERED_RAW_ACCESS based MCP copy utility is not affected by the change at all.

The NEXT change in the CONFIGURATION was to enlarge the read AHEAD to 16 and

CHECK how this change had affected the two cases:

MCP BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
256K	31.4	0.0	3.2	32.4	0.0	3.5

REGULAR BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
1K	1:23	1.2	47.6	1:07	1.1	47.4
256K	57.5	0.0	36.2	57.6	0.0	37.6

There is an improvement in the BUFFERED I/O application and none on the Mcp side. The GAP between the BUFFERED I/O and the UNBUFFERED I/O is closed. One could say that then the UNBUFFERED I/O is actually redundant. The fact is that a configuration like the ONE under this test will do some good to a very specific type of application: ONE which need BIG CHUNKS of CONTIGUOUS DATA, but on a MULTI-USER system there are many other applications which do not have this TYPE of characteristics and they will read 16K BYTES AHEAD while they actually needs much less. For such application this "IMPROVEMENT" won't do any good, but on the contrary, it may slow them down like an IO bounded PROGRAMS.

In order to check whether the performance of the Mcp will tend to decline for BIGGER FILES in comparison to the I/O performance of the BUFFERED I/O a COPY of a 20Mbytes FILE was tested as well (still read AHEAD == 16 and CACHE BUFFER NUMBER is 256)

MCP BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
256K	1:03	0.0	6.2	1:03	0.1	6.6

REGULAR BUFFER SIZE	A ONE DISC COPY REAL	USER	SYS	TWO DISCS COPY REAL	USER	SYS
8K	2:18	0.3	1:14	1:51	0.3	1:14
12K	2:25	0.2	1:13	1:49	0.2	1:14
16K	2:24	0.2	1:15	1:52	0.1	1:17

It seems that the proportions are as they were on the smaller FILE TRANSFER.

The LAST TEST was to TRY BOTH APPLICATIONS under a MULTI USER environment: under this sort of environment there was no guarantee as to the MAX TIME the transfer could take. This was the result the TIME SLICE allocated for each process. If a HIGHER PRIORITY WOULD BE GRANTED to the I/O BOUNDED process, then the result would be about that of the single user environment.

A better performance issue: In the {M|m}cp utility a special calculation is done

in order to MREAD/MWRITE a MULTIPLICATION of the MEDIA BLOCK-SIZE. It is worth going to this trouble because this way the application overcomes an unnecessary BUFFER MECHANISM which would occur if the NUMBER of BYTES would not be around one. An EXTRA READ and WRITE respectively will have to be executed. A FILE is actually a collection of BLOCKS and not just as it might be thought, a collection of BYTES. Therefore it is possible in the Mcp to take this INFORMATION into account and use it for better performance. It is of course THEORETICALLY very helpfull but then when a copy of a FEW MEGA BYTES is the case, this aspect can't influence the total time very much

The mlseek function is supplied to help the user seeking to a certain point in the FILE and from there to conduct a MASS I/O. If too many mlseeks are used, then the UNBUFFERED I/O will be actually mis-used.

An issue which is unrelated to the RAW_UTIL but of some interest: The -A option of the (M|m)cp command. In transfers that involve TWO DISCS, an ASYNC mode can be of much help. There is some SEMAPHOR mechanism available on the HP9000. However it can't send QUANTITIES (LIKE: how much to READ or WRITE). Therefore by using the 'MEMLCK(2)' system call it is possible to implement a sort of ASYNC communication. In order to make it work the BEST, a lot of FINE TUNING is needed. On ONE DISC transfer the results clearly showed that a bigger BUFFER is preferred to an ASYNC mode with two halves of the buffer size.

WRAP UP

The UNBUFFERED RAW ACCESS UTILITY can be USED on the HP9000(500) family.

The package has to be rewritten in order to be used on other UNIX machines.

Programs which use the RAW_UTIL can relatively easily be ported to UNIX MACHINES which do not have the UNBUFFERED_RAW_ACCESS UTILITY.

The UNBUFFERED RAW ACCESS UTILITY introduced a restricted UNBUFFERED I/O mechanism into the UNIX WORLD. (Can't READ/WRITE/LSEEK behind existing FILE END OF FILE).

The UNBUFFERED RAW ACCESS UTILITY is respectfully DEBUGGED, and can be used by the users of the HP9000(500) family. In case of malfunction the AUTHOR is more than willing to hear about it and the reported BUGS will be fixed.

9010. A TEST PROGRAM DEVELOPMENT SYSTEM BASED ON THE HP-9000 FAMILY OF COMPUTERS

Jerry C. Merritt
Technology Development Corporation
Arlington, Texas

John McCutcheon/Robert Atkinson
TRW, Redondo Beach, California

BACKGROUND

Historically, a unique test software and hardware system was often developed to meet specific project requirements. During the past several years, however, existing test sets (one or more racks of test instruments, special purpose drawers, and instrument controllers) have often been modified for use on more than one project, and sometimes modified to test more than one type or class of module. New circuit designs and technology usually require more demanding tests (i.e. higher frequency, lower noise characteristics, etc.) resulting in more sophisticated test instrument requirements. Consequently, this hardware approach had only been moderately effective at reducing costs. New software was also normally written to support the newly modified test set since making major modifications to existing test programs was often found to be very expensive and time consuming. In addition, some new software functionality was usually required to operate new test instruments. As is typical with many low-production, high technology aerospace companies, the advantages of semi-automatic testing (increased throughput, improved repeatability, etc.) could easily be overtaken by the relatively high per-unit cost of developing software.

MATS CONCEPT

The MATS concept was initially formulated within the Hybrid Manufacturing Department (HMD) of the Manufacturing Division at TRW. Most of the original goals of MATS were thus directed toward reducing the costs and improving the performance of testing within the HMD. The primary test target or Unit Under Test (UUT) is typically a medium to complex hybrid circuit used in demanding aerospace environments. The original goals included: reducing test equipment costs; consolidating existing special purpose test sets (20-plus) into three general purpose test sets; developing a generic approach to modular, reusable software functions; and significantly reducing the test software development costs. Later these original goals were expanded to allow use in other manufacturing-related departments. Some of these added goals included: test software configuration management and data base management systems; communications capabilities with other systems (via modem or local area network(LAN)/IEEE 802.3); output spooling for shared printing and plotting requirements; adaptability to any instrument-intensive test set via IEEE-488 bus; complete, easily understood documentation; and a user- friendly, menu-driven executive system.

To facilitate producing a system which could meet these goals, TRW entered into a system development agreement with Technology Development Corporation (TDC), located in Arlington, Texas. After a short evaluation/concept study, three development phases were scheduled as follows. During the first phase, the

overall system design would be completed, as well as completion of the critical Test Program Generator and main operator menus. The second phase would address the file and configuration management system, as well as the lower-level menu system. The data base management system would be completed during the third phase. As of this writing, the first two of these phases have been completed, and the third is underway. Following each of the phases, acceptance tests have been conducted to verify the MATS performance requirements. To date, MATS has been demonstrated to meet or exceed initial program goals.

SYSTEM ARCHITECTURE

MATS is targeted toward the HP 9000 family (series 200 and series 500) computers, including the HP Shared Resource Management (SRM) system and associated software. The hardware system includes individual series 200 workstations (comprising local computer, CRT display, keyboard, and mass storage device(s)) connected to individual test equipment peculiar to the class or type of modules/hybrids designated for that station. It also includes a dedicated SRM controller (also 200 series), shared peripherals, and an HP series 500 computer which is used to provide extended capabilities in the area of test data analysis/management and communications. A block diagram of the prototype MATS hardware configuration is shown in Figure 1.

The HP SRM system involves both hardware and software which, integrated as a central controller, functions both as a general purpose file server and a shared peripheral management service for MATS. This allows MATS software to centralize data base/file activities on a user-oriented basis, and to enable fast and easy access to controlled files. Several key software modules comprise the SRM operating system: a file system, a multi-tasking kernel (allows multiple requests or tasks to be performed at essentially the same time), a dispatcher, a spooler facility, and an operator console. The SRM system currently enables a maximum of 25 users to share discs, printers, and plotters; hardware connection of workstations is via a dedicated HP SRM interface/adapter and standard coax cable with a maximum of 1000 meters between controller and station. Maximum link speed of an SRM system is 700 Kb/sec, which results in an approximate file transfer rate of 15-35 Kb/sec. The main attribute of the SRM system is the file sharing capability available to users through their workstations. The SRM system handles all files through a hierarchical, tree-like structure which allows MATS to logically group them by user or specific project application. The number of directories and subdirectories under each is essentially unlimited, with each file name allowed up to 16 characters, and a date/time stamp automatically appended to each file. MATS takes full advantage of these attributes to provide a customized but controlled-access interface to the SRM system.

SYSTEM SOFTWARE DESIGN

MATS software is based upon the HP BASIC Operating System (version 3.0) on workstations and the SRM Operating System (version 2.0) on the SRM controller. This allows full support of all SRM and hardware capabilities as well as ease of adaptability for the typical test engineer. Though the software environment is controlled from power-on, access to the BASIC system is available for applications external to MATS and for new test function and instrument driver development. MATS software is comprised of three major components controlled by a single executive component as follows:

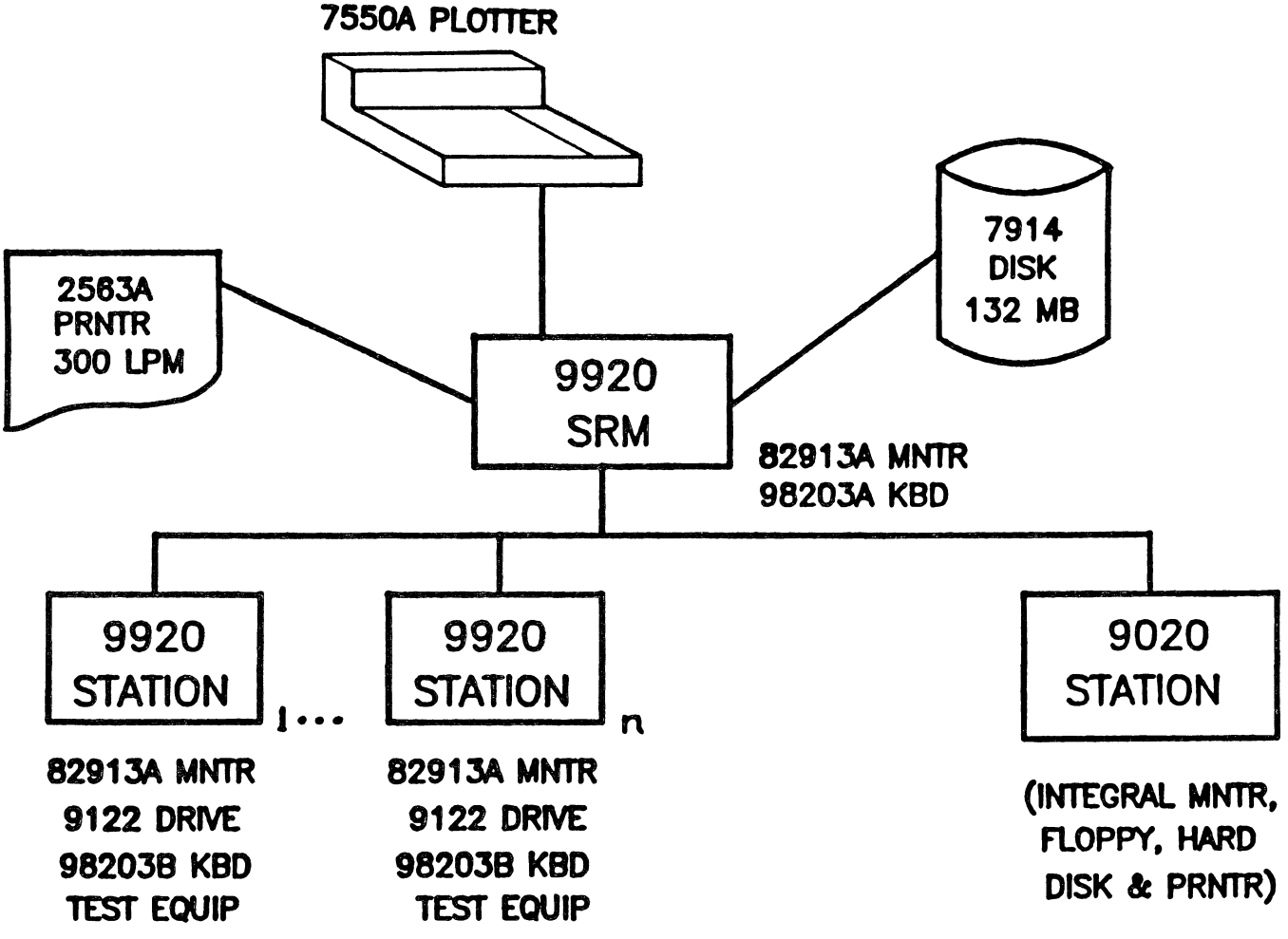


Figure 1 — System Hardware Configuration

1) MATS Control - main executive controlling access to the software functions and their components

2) Test Program Generator - software to support all phases of creating an executable Test Program targeted for a particular UUT and associated test station configuration

3) Test Program Executive - control software to support initiating, loading, and running an existing Test Program in the MATS environment, and properly routing output data to the appropriate files and devices

4) Data and Control Manager - system software functions used to manipulate and transfer data/files, modify user or test station capabilities/parameters, and perform supplementary configuration management and control functions

A block diagram of the MATS software and its primary interfaces is shown in Figure 2. MATS software is designed for use on one or more of the host hardware test stations (series 200 computer) or the series 500 computer. The latter is designated the data analysis/communication computer, and provides for expansion of the data base and communication- related capabilities of the basic MATS system. MATS provides access to its lower level functions through a series of interactive terminal screen selections made by the user, using the keyboard function (soft) keys as the primary access mechanism. On-line help for use of the functions is generally available in the form of text displays describing user options when requested by the operator. Software interfaces between major MATS components are effected through BASIC common blocks, passed parameters, and the MATS data base file structure.

The MATS Control function performs overall MATS executive functions, including system access (via password controlled logon/logoff), general system initialization and termination, and system-level error processing not handled by other subcomponents. It performs user set-up and identification for new users, validates MATS user privileges, and provides top level menu screens as a mechanism to invoke lower level functions. Upon completion of MATS functions, it accomodates termination of MATS activity, including any required file closure and resource deallocation processing. It also allows straightforward access to the BASIC 3.0 operating system environment. All MATS Control functions are invoked from a resident protected file on the SRM disk. The command screen at the highest level of the MATS hierarchy is shown in Figure 3.

The Test Program Generator enables the user to create new Test Programs through a process of parameterizing and assembling a set of user-defined Test Functions and related parameter files into an integrated sequence of tests. Individual Test Functions are general-purpose subprograms dedicated to performing a specific class or type of test on a circuit to evaluate its operational performance under controlled conditions. Examples would include power application functions, frequency measurements, phase/gain tests, power output tests, and current-voltage measurements. Test Functions typically call one or more Instrument Drivers (also subprograms) which translate Test Function stimulus, response, commands and data into instrument-specific codes for processing by a target programmable test instrument (typically via the IEEE-488 bus). The Test Program Generator allows the user to select any number of these Test Functions and to select

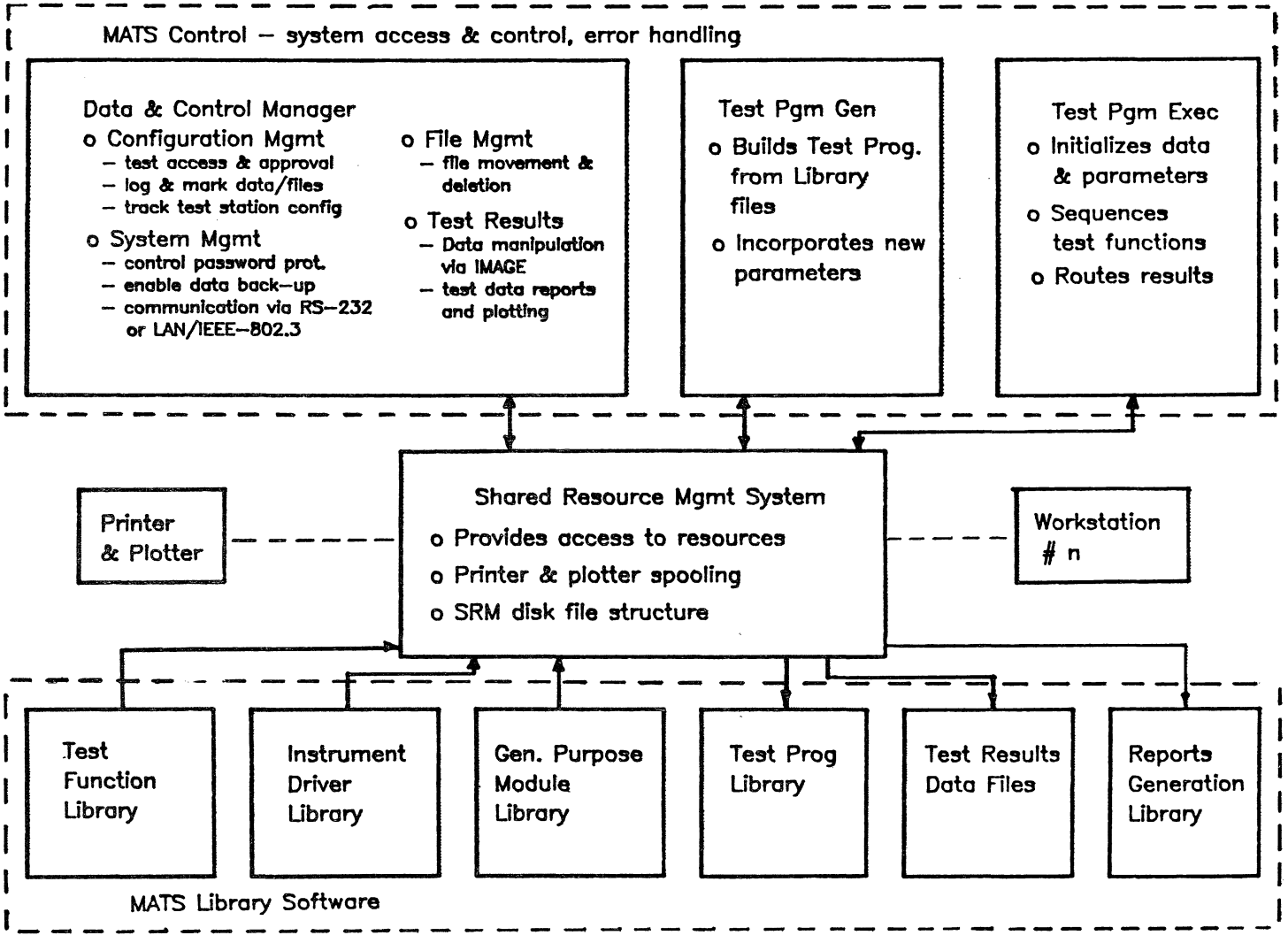


Figure 2 – MATS Software Configuration

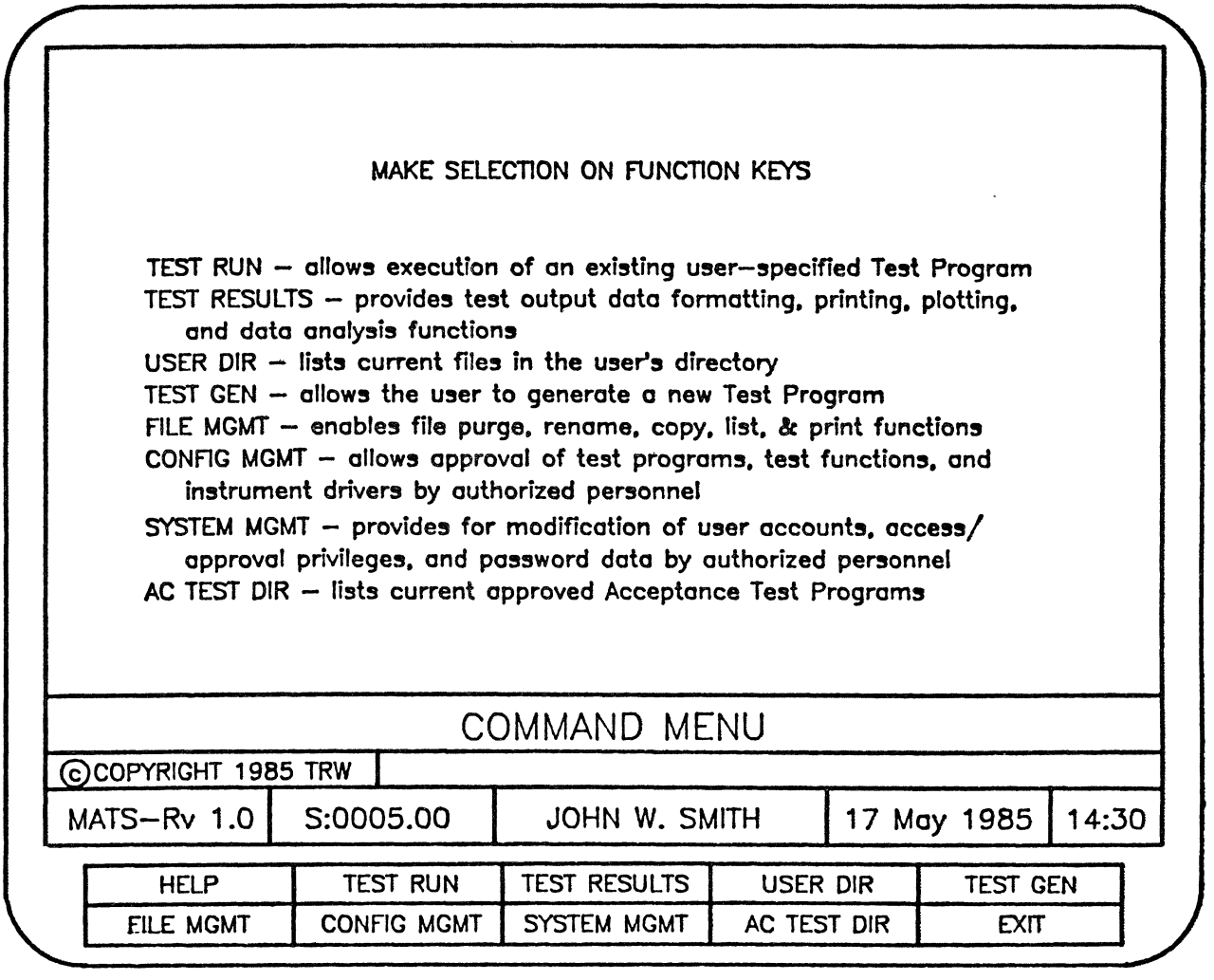


Figure 3 – MATS Command Menu

parameter values which customize the function for his particular use. Also provided are menus which allow identification of calibration functions/outputs, classification of several types of test runs (i.e. functional test, pre- seal test, burn-in test, etc.), and identification of test specification data (files) which designate supplementary test data, such as upper/lower limit values for tests. Once all Test Functions and related data have been identified through an interactive session, this information is saved as an intermediate file; it may then be later modified or processed by the Test Program Generator by 'assembling' the source data into a completely integrated Test Program. The result is a uniformly constructed sequence of functions and tests wrapped within the Test Executive 'shell' which provides a common user interface at run time.

The Test Program Executive provides a controlled execution environment for MATS by performing common initialization and run-time options for each Test Program run, and establishing formatted data output files and controls for error-handling. This includes validation of selected test station configuration (test instruments), selection of which Test Functions are to be executed (user-selected or based on test run class), and sequencing of test functions. Once all test resources and options have been validated, Test Program execution is primarily dictated by Test Function code. Should fatal errors occur which are not detected by the Test Program, the Test Program Executive has the capability to terminate the execution process. As a separate function, the Executive provides test output formatting and routing to appropriate peripheral devices. This includes user selectable options for final disposition of test results.

The Data and Control Manager enables configuration control and tracking of all MATS-resident Test Programs, Test Functions, and Instrument Drivers through a simple set of screen prompts and password validations. Files submitted for a formal approval cycle are software stamped such that engineering approval levels can be readily observed/printed without compromising file security. A status control file is maintained for each submitted file which documents the approval 'signatures' and related control information. When formal approval has been completed, MATS allows an authorized user to enable transfer of the file to a permanent library-type directory from which other MATS users may draw on for their own needs. Data and Control Manager software also provides a complete set of functions for file manipulation (i.e. copy, rename, purge, list/print), system management (MATS user modification and directory control), and station configuration management. This latter capability enables a user to define and track specific workstation test instrument assets as a controlled entity (by instrument name, bus address) so that the integrity of hardware required by the Test Program is ensured. In the case of the series 500 computer, a Data and Control Manager function is provided which supports communication with computers external to the MATS hardware system (via RS-232 link and later a LAN), as well as supplementary data analysis subprograms and utilities which may be used to further manipulate output test data. Provision is made for incorporation of HP IMAGE software to enable comprehensive data base management capabilities for the test results files.

CURRENT STATUS AND LIMITATIONS

MATS was originally designed and developed for the Hybrid Manufacturing Department at TRW in the 1984-1985 time frame. The current MATS system was successfully demonstrated in April, 1985 and has been informally released for use as version

1.0. At the time of this writing it is being used for the first time in an actual engineering and manufacturing environment at TRW for a specific hybrid test application. The Test Function and Instrument Driver libraries are being established through requirements for testing these and other planned hybrid circuits currently in the design/production phase. MATS software has been put under configuration control so that problems/enhancements which are identified may be duly addressed and considered for later correction and implementation.

Some current limitations have surfaced with the initial MATS release related primarily to the Test Program generation process. Due to the standard but somewhat sizable overhead folded into each Test Program generated, minimum program size is fixed (approx. 250KB), even for the most trivial Test Programs. Likewise, execution loading and initialization may be somewhat slower than could be achieved with a completely customized test package. A planned increase in standard workstation main memory (from 1 to 2MB) will minimize these constraints. Since there were few control features extended to the user for program/ logic control during program execution, it may be observed that special test cases requiring varying and unique looping or conditional branching constructs at run time can be inconvenient to implement. Currently these may be handled through manual modification of the generated Test Program source file, but it is planned that special control functions will be implemented in the next release of MATS. Finally, due to a relatively low priority placed on the communications and DBMS features initially planned for the original release, the associated file transfer and output data base manipulation functions of the software have not been fully integrated with the system as yet. These also are intended to be implemented in a future version of MATS, probably in the fall 1985 time period.

CONCLUSIONS

The major goals of reducing test software costs and reducing skill levels required to generate test programs have been met with substantial success with the initial release of the MATS system. Cost savings are realized primarily by accelerating the Test Program generation process through automated assembly of Test Functions and Instrument Drivers into an integrated whole, and by implementation of a series of stored, reusable, generic functions and drivers contained in protected libraries. By the nature of this system, as time progresses and more functions and drivers are developed, this cost savings will continue to increase. Required user skill levels have been reduced by enabling minimally trained personnel to access these libraries, and configure new test programs essentially by modifying UUT or test-specific parameters only. Judicious use of the available on-line HELP utility, along with a tailored MATS introductory training course and user's manuals have provided key tools for adapting to MATS. Closer tracking and control of all test software is possible with the Configuration Management functions of the system, thus saving much overhead in test-related paperwork procedures. The primary turning point for implementation of such a system relies on the enforcement of design and coding standards for newly developed Test Functions, and Instrument Drivers. This has been accomplished in the prototype system via a published set of programming style guides which define general test concepts as well as MATS-specific interface requirements.

The success of MATS, as with any software system, relies on its use and acceptance as a productive and efficient tool which can be used not only to reduce costs, but to increase test programmer productivity as well. The implementation of

the reusable Test Function/Instrument Driver concept is a cornerstone of this system, and is considered critical to MATS acceptance. Recent industry studies [1] have indicated that software reusability in general may provide a programming productivity improvement anywhere from 40% to an entire order of magnitude, especially when the total software life-cycle costs are considered. With today's exploding costs associated with hardware and (especially) software development, tools such as MATS may be considered to be the next logical step in enhancing the manufacturing test application environment.

MATS was designed to be a dynamic system. It is based on guiding philosophies which permit change over time. Consequently, we have taken a structured, modular approach to the MATS executive software to allow incorporation of future innovations and enhancements. Of course, because we are working to 'real' needs and budgets, some compromises had to be made along the way. But, one of the fundamental concepts we have tried to follow is to allow for future growth and change. This philosophy and the resultant planning will give MATS a long and useful life.

REFERENCES

1. E. Horowitz and J.B. Munson, "An Expansive View of Reusable Software", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September, 1984

9011. INTEGRATING MULTIPLE PROGRAMS UNDER HP-UX

Gary D. Anderson and Robert S. Sciuk
Statistical Software Group
204,200 James St. South
Hamilton Ont. L8P 3A9

The Application Setting:

The Statistical Software Group (SSG), acts as a converter and distributor to HP computer users of standard data analysis software packages, traditionally available on 32 bit mainframe systems. Since its inception in 1977, the SSG has converted and maintained the BMDP, SPSS, Minitab and SCSS packages on the HP3000 and, more recently, STAT80 on the HP3000. Since 1983, the SSG has also been involved in converting and distributing BMDP-UX, SPSS-X, Minitab, STAT80 and SIR on HP-UX systems.

BMDP provides a good example of a very powerful batch oriented package that uniquely meets the demands of large data analysis tasks. To use this package effectively in an on-line 'workstation' environment, the user needs easy movement to and from an editor. Sets of commands must be entered, submitted and then easily modified and re-submitted for each new analysis step. The user also needs a good facility for working with the output from the analysis steps. Often only small parts of the output stream from a data analysis session need to be printed. A facility for quickly browsing through the output to find the important parts and for easily moving those parts to a file or printer is a requirement.

This paper describes a project, undertaken by SSG, to integrate an interactive front-end system with the BMDP package under HP-UX. The combined statistical package and front-end system has been named BMDP-UX. BMDP-UX provides an example of the type of interactive system that can be produced by forcing a batch program to communicate with interactive driver programs through appropriate interprocess communication channels.

The BMDP Statistical Package

The BMDP statistical package is the oldest and most highly respected statistical analysis system in existence. It is a direct descendant of the BMD Biomedical Computer Programs first published at UCLA in 1961. The package has continued to be enhanced and expanded over the years. Today, BMDP is used widely around the world and is recognized as a statistical standard against which other systems are measured. The present BMDP package is a set of 42 very comprehensive statistical analysis programs. The programs are loosely classified into eight series as follows:

- D: data description
- F: frequency tables analysis
- R: regression analysis
- V: analysis of variance
- M: multivariate analysis
- L: life tables and survival analysis

S: non-parametric analysis
T: time series analysis

The BMDP package features a common sentence, paragraph structured command language as well as a common binary save file which can be written to, or read from, by any of the 42 programs in the package, interchangeably. The common BMDP language is used by all of the programs for describing the data to be analyzed and for requesting analysis and display options. The BMDP programs are ideally suited for use with an interactive front end which integrates access to the many separate programs into a unified system.

The work described in this paper is based on an interactive system, developed by BMDP Inc., for use on a dedicated UNIX workstation called the STAT-CAT. BMDP-UX is an extension of the original STAT-CAT system and has been modified and extended for the HP-UX environment (HP9000 series 500, HP9000 series 200 and the Integral PC).

Features of the Interactive Interface:

The most prominent feature of the BMDP-UX interactive interface is a screen 'window' which is maintained by the system over the session input and output stream. This window features a one line, highlighted header banner which continuously displays the name of the current program being executed, the current date and the time of day at the top of the screen. This header serves to keep the user informed, at all times, about what activity is currently running regardless of what may actually be positioned in the window.

The BMDP-UX window, by default, displays the activity happening at the moment, with input commands and program output scrolling from the bottom of the window and out of the top. This takes place in a manner consistent with the use of any standard terminal for interactive processing. At any point at which the system prompts for input, and, at regular points where the executing program has been forced to pause during generation of its output, the window may be moved up, down, left or right for viewing any part of the entire session output stream accumulated to that point. Alternatively, any HP-UX command may be executed at system prompts and pauses.

To facilitate positioning the window over the desired place in the output stream, the arrow keys and the user menu keys associated with the eight highlighted boxes, situated at the bottom of HP terminals, are brought into play. The arrow keys move the window up and down or left and right over the output, in single page, or finer increments. The four right most user menu keys (f4 - f8) are associated with top, bottom, find and mark respectively. Pressing top or bottom moves the window to the beginning or to the end of the current session output stream, respectively. Find and mark allow the user to either position the top of the window at a specified string searched for in the output stream or to mark a location in the output stream with a user specified string marker for later recall. While the user is moving the window over the output stream, the BMDP program is maintained in a pause or 'locked' state. Analysis cannot continue until the window is returned to the bottom of the output stream. At that time, the program is unlocked and execution is allowed to continue.

Once the user has positioned the window over an interesting section in the output stream, he will often want to move that part of the output to a separate file or to a printer. The user menu key f4 is associated with a submenu featuring a set of seven output options. These options include provisions for assigning or reassigning an output file or printer and for writing the present BMDP window contents, any marked section of the output stream or the entire output stream to this assigned file or printer. Using these options the user can easily build one or more output files containing selected parts of the present session output stream.

The remaining main user menu keys (f1 - f3) provide commands for invoking the vi HP-UX system editor to build or edit a BMDP command file or for causing an existing BMDP command file to be executed.

In addition to the functions described, the BMDP-UX system provides additional, less commonly used commands, through CTRL keys. CTRL key alternatives for all user menu commands are also provided for use on non-HP terminals or older HP terminals without user menus.

The Concurrent Programs Integrated in BMDP-UX:

The interactive environment for BMDP-UX is created by three programs working together to process the users requests for data analysis using BMDP. These three programs are the Input Manager, the Output Manager and a BMDP program respectively. The three programs run concurrently and communicate via HP-UX facilities for inter-process communication described in the next section.

The three programs are described as follows:

1. The Input Manager (IPM):

   This program is the first to be started on executing the BMDP-UX command. The Input Manager processes all input from the user. Instructions entered by the user from the terminal are looked at by the Input Manager and passed on to either the BMDP program or the Output Manager for processing. The Input Managers task is to decide which of these two programs the users command is intended for. In general, instructions and data for the BMDP program are terminated by an end-of-line terminator, while Output Manager commands are terminated by a control character, a function key or an arrow key.

   BMDP program commands or data may be entered only when the program is prompting for them. Output Manager commands may be given any time that the program is waiting for the user to type at the keyboard. These times include the BMDP program prompting for input and the program pausing between "pages" of output.

2. The Output Manager (OPM):

   The Output Manager program keeps track of all output from the BMDP program. In an interactive environment, the user normally views the BMDP program output on the screen. The user needs the ability to view any output which has moved off the top of the screen. It is the job of the Output

Manager to process the user's requests to display specific portions of the output stream on the terminal screen or to direct portions to a printer or file.

3. The BMDP program:

   The BMDP program takes input consisting of instructions or data from the user and produces output in the form of reports, lists, charts, tables, files, etc. When being run under the control of the interactive Input Manager, a BMDP program will pause at appropriate points and wait for a user generated indication (<CR>) to continue. The BMDP programs can also be run, independently of the BMDP-UX interactive system, as stand alone batch programs. When running in batch mode, a BMDP program will run to the completion of its instructions and produce all appropriate output without pausing.

Access to the vi editor and to HP-UX operating system commands is achieved by the Input Manager detecting an ESCAPE as the first character of a command which signals that the command is to be passed directly to HP-UX. The editor is executed in its own shell. The editor is invoked in BMDP-UX by pressing the user menu key f3. The editor uses a default 'input scratchpad' file. The current BMDP instruction set is maintained in this input file. Upon leaving the editor, the scratchpad file is automatically re- written and re-submitted to the BMDP program currently running. Thus, a user can repeatedly modify and re-submit a BMDP instruction set by merely pressing the user menu key f3.

In addition to the input scratchpad file, which contains the most recent instruction set submitted to the BMDP program, the Input Manager also maintains a log file. This file contains a complete chronological log of all commands and instruction sets submitted. By editing this log file, a user can recapture any instruction set or command sequence used, up to that point, in a BMDP data analysis session.

To cause the editor to access a file other than the default input scratchpad file, the user merely types the name of the desired file before pressing the user menu key f3. The same method is used to specify files other than the default for all commands involving a file.

Process Synchronization and Communication:

To address the problem of synchronization and flow of information between the asynchronous processes which make up the BMDP-UX environment, a system of "signals" and "pipes" was used. Although HP-UX version 5.0 is expected to support the interprocess communication facilities of AT&T Unix System V, at the time of this writing semaphores and shared memory are "unsupported" in the HP-UX version 4.02 that we are currently running. A discussion of the implementation and use of semaphores can be found in Using Interprocess Communication to Implement Database Concurrency Under HP-UX which appears elsewhere in these proceedings.

For those not familiar with the concepts of signals and pipes in Unix, a very good description is given in The Unix Program Environment by Kernighan and Pike.

Signals:

Signals are sent to all processes initiated from the same terminal by events such as interrupts, software errors, hanging up a modem, or user intervention (eg: striking the DEL key on the terminal keyboard). Usually, a signal sent to a process will cause that process to terminate, unless that particular signal is caught and handled by specific code to either handle or ignore that signal. Signals may be trapped by application software through the "signal" system call which takes as its arguments a signal number (there are 19) and the address of a user-defined routine to handle that signal. One process may signal another process specifically by means of the "kill" system call which requires a process identification number (pid) and a signal number. The signaled process will immediately branch to the code specified to handle that particular signal.

Pipes:

Pipes provide a mechanism for connecting processes in a producer-consumer relationship, where one process reads as input the output of another process directly (ie: no file I/O is required). A pipe is an in-core first-in-first-out (FIFO) serial buffer, and may be either named or unnamed. Synchronization of pipe reads and writes is done automatically by the HP-UX kernel through the internal system calls "sleep" and "wakeup". One process may invoke another process by means of the "fork" and "exec" system calls and create a two way communication channel between them by means of the "pipe" system call. Note that although the "popen" system call combines the fork, exec and pipe calls all into one convenient system call, the child process' pid which is normally returned from the fork call and is used in the kill system call (see above) is not returned from popen. For this reason the fork - exec - pipe sequence was used.

Filters:

The concepts of filters and pipes are fundamental to the Unix environment. Unix consists of a number of simple programs which perform simple tasks, but which may also be combined in various ways to effect very elegant results. The only requirement for a program to be a filter is that it read from standard input, and write to standard output. As an example, the following command sequence will search through the file "myfile" for lines containing the string "Error", append those lines to a file called "errorfile" and display a count of such lines on the terminal.

```
grep "Error" myfile | tee -a errorfile | wc -l
```

The "|" or pipe symbol means connect the standard output of the first program to the standard input of the second program - a sort of in-line connector. The "tee" program provides a T-join into a pipe, to direct the flow into two directions at once. Thus the analogy to plumbing or pipe-fitting. Much more complex arrangements can be made between programs as will be demonstrated below.

The Plumbers Toolbox:

An example of the code used to create a process, and to open a pipe between them is given below. For more detail of the actual pipe fitting the reader is referred

to The Unix Program Environment and the HP-UX Reference Manual.

First, a pipe is opened. Opmfdl is an integer array of 2 elements containing "file descriptors" for the read (0) and the write (1) ends of the pipe.

```
if (pipe (opmfdl) == SYS_ERR)    /* open a pipe */
        fatal ("couldn't get opm pipes");
```

Fork will create an exact duplicate of the current process, which will immediately begin execution. The son process id will be returned from fork and stored in opmid in the parent process, and 0 will be returned in the child process. Since it can be determined whether the code executing is the parent or child, the code below will be executed only in the child process.

```
if ((opmid = fork ()) == 0) {/* Fork a process */
        close (0);           /* Close stdin     */
        dup (opmfdl[0]);     /* Attach pipe to stdin (read) */
        close (opmfdl[1]);   /* Discard write side of pipe */
        execlp (opm,opm,0);  /* Execute opm */
```

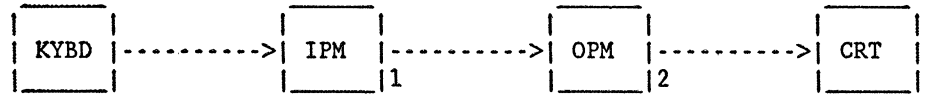
The "execlp" system call will overlay the current (child) process with the OPM program, and there can be no return from this call. The "close" system call will close a file, and takes as its argument a file descriptor. By convention the file descriptors 0, 1 and 2 are standard input, standard output and standard error respectively. The "dup" system call will force its argument (a file descriptor to an open file) to be connected to the lowest available descriptor. Since we have just closed 0 (stdin) the read side of the pipe will be connected as standard input to the child. Since the standard files are inherited across an execlp call, OPM will be connected to the pipe when it overlays the child copy of IPM.

In the parent process, the write side of the pipe is attached to standard output in a similar fashion, and the read side of the pipe is discarded.

```
close (1);          /* Close stdout */
dup (opmfdl[1]);    /* Attach pipe to stdout (write) */
close (opmfdl[0]);/* Throw away read side of pipe */
```

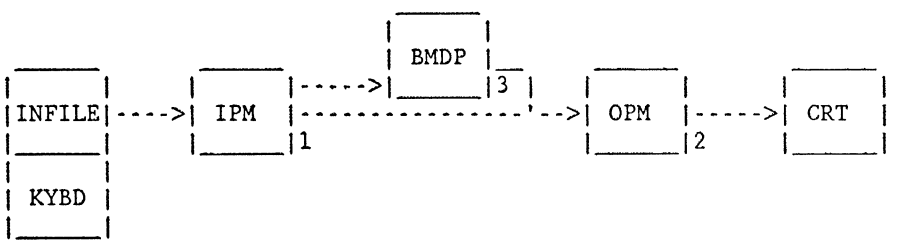
The Plumbing:

At initiation, the BMDP-UX environment can be depicted simply as follows:



The IPM will invoke the OPM as described in the section above. OPM will write its standard output to the terminal. At this point, the user will be in the BMDP-UX environment and is able to initiate any HP-UX command, specify that a

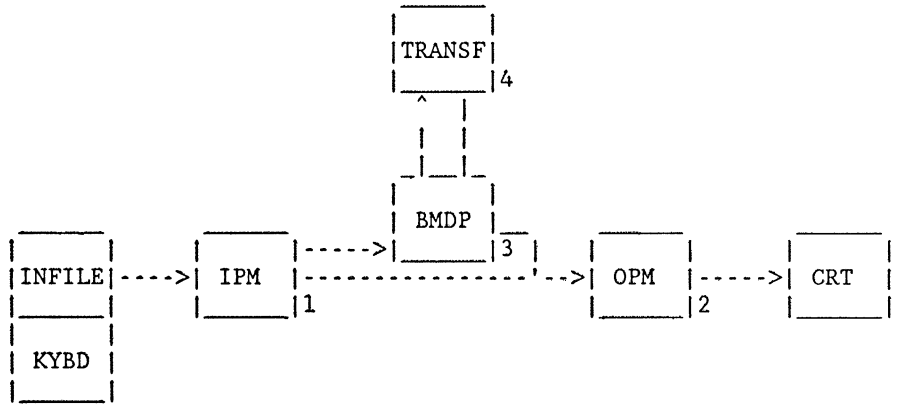
BMDP analysis be done, or review any ASCII files. If the user invokes a BMDP program, the following situation arises:



The above diagram indicates that a scratch INFILE now provides input to IPM, and a new pipe is created for the BMDP process. The input to IPM is handled like a stack (LIFO), and the input scratch file is checked periodically, to determine whether modifications have been made to it. If this is the case, and BMDP is expecting input, the input scratch file will be made the input file.

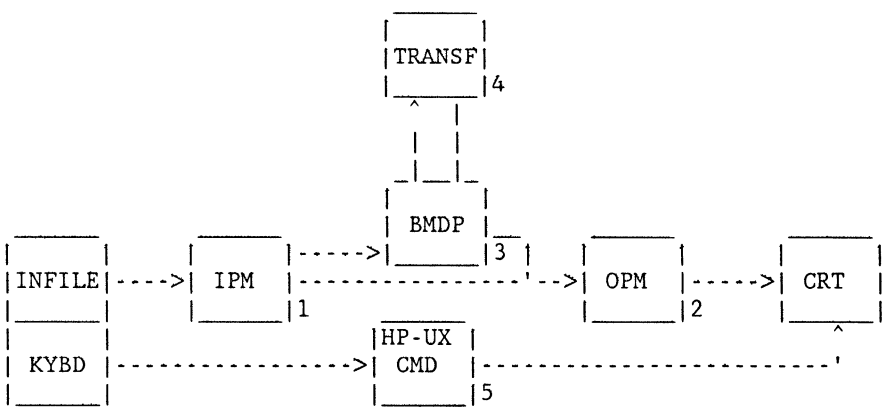
It is interesting to note that the standard output of the BMDP process and the standard output of the IPM process are joined so that the OPM process is reading information from both the IPM (eg: file editing control sequences) and BMDP (eg: results of the analysis of the input data) on its standard input file. It is a result of this situation that synchronization of the several processes of the BMDP-UX environment becomes necessary.

At this point it may be useful to indicate the manner in which BMDP-UX performs its transformation of incoming data. If the user had indicated that user defined transformations were to take place before the analysis was performed, a stand alone "filter" would be added to the processes as follows:



The transformation program can be any C, FORTRAN, Pascal etc. program that reads input from its standard input, and writes its output to standard output in a specified manner (ie: a Unix "filter").

Finally, if the user invokes an HP-UX command (eg: 'vi') while running a BMDP program with a transformation function, the following situation will arise:



In the case of the HP-UX command, the generated output, if any, will be displayed, but not buffered by the OPM. Any required files for the HP-UX command will be opened as needed.

Keeping Signals Straight:

OPM must keep screen handling control sequences sent from the IPM separate from the output of BMDP programs. In order to do this, BMDP must signal IPM when it pauses periodically to allow commands from the keyboard to be sent to OPM. Upon invoking a BMDP program, the IPM will enter the "wait_for_bmdp" procedure which consists of the following C code:

```
wait_for_bmdp()
 {
     if ( bmdp_done )          /* Check flag set by interrupt */
             return (true);    /* If true - return */
      else
             pause();          /* Sleep until signaled */
  }
```

An interrupt handling routine will trap the signal sent from the BMDP program to the IPM, and set the bmdp_done flag (a global variable). The "pause" system command will cause the calling process (IPM) to sleep until awakened by a signal from the BMDP program. Upon closer examination, the reader will notice a slight problem with the above code.

Critical Regions:

There exists a critical region between the test condition and the sleep operation. If the signal from BMDP happens to arrive just after the test condition and just prior to the "pause" system command, the wakeup signal will be missed by the IPM, which will then go to sleep never to re-awaken. Since the kernel handles the synchronization over a pipe and presumably at this point the BMDP process

is sleeping on an empty pipe, deadlock will result as both processes wait for signals that can never come.

As the code generated from the above C source will consist of several machine instructions, there is a small but finite possibility of deadlock occurring. Heavy loading of the system and insufficient physical memory may increase the probability of deadlock. The problem could be resolved by a system primitive which would guarantee an uninterruptible (atomic) test-and-sleep operation.

If deadlock does occur, the kill signal may be generated from the keyboard (cntrl-\) and the opm will begin an orderly shutdown of the environment, saving the output buffer to disc, and closing log and input files. The user may then re-enter the BMDP-UX environment and retrieve the entire saved output buffer to continue processing.

Validity:

Although the "signal" and "pipe" method of interprocess communication is not as robust as the "semaphore" and "shared memory" method outlined elsewhere in these proceedings, several advantages exist to this approach. Simple programs which need only read standard input and write standard output (filters) can be combined in various ways to produce sophisticated results with very little development overhead. With the "test-and-sleep" primitive described above, signals would become more robust in their ability to synchronize multiple processes.

In light of the above discussion, and the imminent availability of the IPC facilities of HP-UX version 5.0 it is felt that this approach for the BMDP-UX environment is valid for the interim period.

Summary and Conclusions:

BMDP-UX provides an illustration of the degree to which an interactive environment can be created around a traditionally batch system. The authors feel that the resulting environment provides a very powerful and user efficient data analysis system under HP-UX. We consider BMDP-UX to be a prototype implementation of a more extensive interactive front-end system.

Because of the relative independence of the Input Manager, the Output Manager and the vi editor from the BMDP programs themselves, it is felt that a more general interactive interface system should be developed which will front end to other batch systems. The SSG plans to extend the functionality of the Output Manager and to include access to other statistical packages, graphical systems and possibly a word processor from within the system. The degree to which this activity is carried will depend, to a great extent on the market acceptance of the initial BMDP-UX implementation described in this paper.

9012. PERFORMANCE OF MULTIPLE PROCESSORS IN THE HP9000 SERIES 500

Robert J. Bury
Hewlett Packard Co.
Fort Collins System Division
3404 East Harmony Road
Fort Collins, Colorado 80525

I. Introduction

Multiple processor architectures can provide many attractive benefits. In particular multiple processor systems offer the potential for greater system performance and throughput. Modular multiple processors can also provide a convenient and cost effective means of expanding the capabilities of an existing system. The potential of multiple processor systems, however, can only be realized through careful design of both hardware and software.

The HP 9000 Series 500 is a tightly coupled multiple processor machine that can be configured with one, two or three CPUs in the same system. The hardware architecture and implementation supports multiprocessing with a high bandwidth to globally shared memory, fundamental synchronization mechanisms, and a symmetric approach to I/O. The system software complements the multiprocessing hardware with a reentrant kernel that fully supports symmetric multiprocessing. Multiple tasks can simultaneously execute within the operating system kernel. The overall design results in an efficient multiple processor system that is totally transparent to application programs.

The granularity of multiprocessing on the HP 9000 Series 500 is at the task level. This is very well suited to the multitasking nature of the HP-UX operating system. Single users can experience performance benefits from multiprocessing by concurrently executing multiple tasks. Some single programs can achieve performance gains from multiprocessing if they are decomposed into separate tasks that coordinate through inter-process communication techniques. The actual incremental performance gained from multiple processors is dependent on the characteristics of the system load.

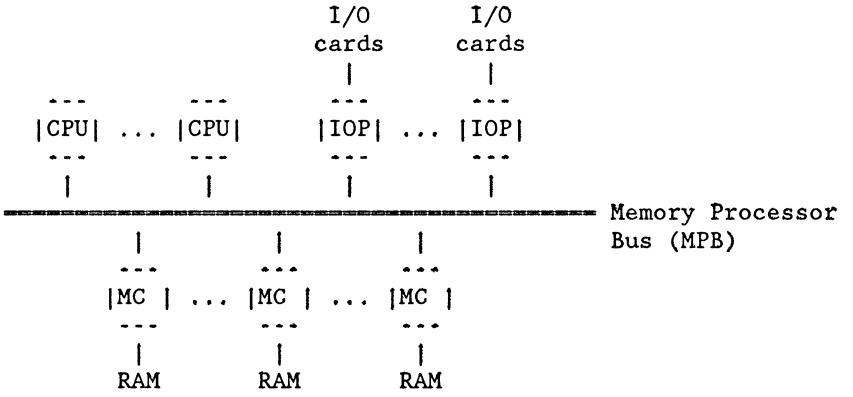
Multiple processors add another dimension of configurability to a computer system. An optimum configuration should exhibit a proper balance of processor, memory and I/O resources that is well matched to the intended application and system load. A system monitoring tool can be used to dynamically measure the performance and behavior of the system in order to understand the effects of different hardware configurations.

II. Hardware Support of Multiprocessing

The HP 9000 Series 500 hardware was designed as a multiprocessor system from very near its inception. As a result, the hardware support for multiprocessing is an integral part of the architecture. It was not added on as an afterthought.

The architecture of the HP 9000 Series 500 can be described as a tightly coupled symmetric multiprocessor with globally shared memory equally accessible by all

processors. A system is composed of three basic modules: CPUs, I/O Processors (IOPs), and Memory boards, which consist of a Memory Controller (MC) and a RAM array. These three types of component boards communicate over a single shared bus. A high level diagram of the basic architecture can be seen in the following diagram.



Any successful multiprocessor system must provide an adequate bandwidth to memory for all processors. The HP 9000 Series 500 accomplishes this with a high speed Memory Processor Bus capable of transferring data at 36 Megabytes/second. This high bandwidth bus is capable of supplying the data and instruction demands of three parallel processors executing typical code sequences, and still deliver bandwidth for I/O DMAs. An arbitration scheme on the Memory Processor Bus ensures that no processor can be indefinitely denied access to the bus. The prevention of processor starvation is very important in a multiprocessor system in order to avoid system deadlock.

One very important aspect of the HP 9000 Series 500 architecture is its symmetry. This symmetry is key to an efficient multiprocessor implementation. All I/O devices are equally accessible by all processors. All processors are equally capable of performing I/O to any I/O device. Of equal importance is the fact that I/O devices can request service by any processor in the system. This allows a task to initiate an I/O transaction while executing on one CPU, for the interrupt to be serviced by another CPU, and for the task to continue execution on a third CPU. This feature of the architecture is essential to building a system in which the operating system can execute transparently and in parallel on all processors.

In addition, there is no hierarchy or master-slave relationships between processors, except for some power-on initialization functions. After power-on, all processors are identical and equally capable of independently executing tasks.

The Memory Controllers provide the fundamental synchronization mechanism to synchronize processors. In addition to the normal read and write memory operations, the Memory Controllers implement a semaphore-load operation, which indivisibly reads the contents of the addressed location and writes a known lock pattern (-1) into that location. This atomic memory operation is used as the fundamental building block of all system synchronization mechanisms.

The HP 9000 Series 500 architecture supports memory management and virtual memory through microcoded address translation using segment tables and page tables. These globally addressable tables can be concurrently accessed by multiple processors. The tables can be referenced by both the processor microcode and by the operating system's memory manager. The processor microcode synchronizes access to these global tables and defines a synchronization mechanism that allows the operating system to synchronize with the microcode and with itself. In this way, the integrity of these shared global tables is guaranteed in the face of concurrent access by the microcode and operating system executing in parallel on multiple processors.

The HP 9000 Series 500 CPU microcode provides hardware support for many typical operating systems functions, including process switching, linked list operations, and semaphore synchronization operations. While these hardware features are beneficial to most any operating system, they are especially important to a multiprocessor in order to minimize system software overhead.

III. System Software Support for Multiprocessing

An efficient multiprocessor system depends not only on adequate hardware support, but also on the careful design of the operating system. A poorly designed operating system can negate nearly all of the performance benefits of a multiprocessor by introducing bottlenecks or by requiring the user to contort his application program in order to accommodate the requirements of the multiprocessor.

A multiprocessor HP 9000 Series 500 can execute either the HP-UX operating system or the HP BASIC language processing system. Both of these operating systems make very efficient use of multiple processors with a minimum of bottlenecks. In addition, the multiprocessing is completely transparent to all application programs, with the obvious exception that a set of tasks will execute faster on a system with more processors. No changes of any kind are required to transport a program from a single processor system to a multiprocessor. Existing programs can be moved from a single processor system to a multiprocessor with no recompilation. In fact the exact same operating system is used regardless of the number of processors installed in the system.

The granularity of multiprocessing parallelism on the HP 9000 Series 500 is the task, or process. In this sense, granularity is a measure of the quanta of work that can be performed in parallel by the system. In the HP 9000 Series 500 system, multiple tasks can truly execute in parallel on multiple processors, but a single task can execute only on one processor regardless of the number available.

The HP 9000 Series 500 operating systems dispatch runnable tasks onto available processors. In this way, processors are treated as resources to be allocated to the most deserving tasks, as defined by the scheduling algorithms. Each processor is responsible for selecting the task to be executed on that processor at any given time. All tasks are maintained on one global task queue which is serviced by all available processors. All processors attempt to find a new task to execute whenever the currently executing task becomes logically blocked (as when waiting for I/O), or when the current task's time slice expires, or when the processor is idle. At these times each processor searches the global task

queue, with the required interprocessor synchronization, looking for the highest priority runnable task that is not already executing on another processor. When one is found, that task is dispatched to the searching processor and other processors are then allowed to examine the global task queue in search of other runnable tasks.

If no runnable task can be found, then the processor will enter an idle state, waiting for any event that might make a task runnable. With the help of a microcoded instruction, an idle processor consumes no Memory Processor Bus cycles. By avoiding a busy-wait idle loop, a system with multiple processors can always execute a single task as least as fast as a single processor system. This is important to ensure that multiple processors are never a hindrance to performance, even in the corner case of having only one runnable task in the system.

The HP 9000 Series 500 operating systems are completely reentrant. This allows the operating systems to be concurrently executed on multiple processors. This is essential to efficient multiprocessor performance since it allows an application program to freely call into the kernel and execute operating systems services without regard to the processor that it is executing on and without waiting for other processors to complete execution in the kernel. Synchronization mechanisms are distributed throughout the kernel to protect critical sections of code from parallel execution on multiple processors. The distributed nature of these synchronization points avoids the bottleneck that would result if all kernel calls caused the serialization of processors. By separating and distributing the synchronization mechanisms, only processors that concurrently demand the same resource will serialize. Tasks that require resources that are currently in use by other tasks will block until that resource is available, freeing up the executing processor to dispatch another task. The synchronization operations are also supported by the processor hardware to achieve fast performance and low overhead.

The kernel is also preemptable, allowing a higher priority task to interrupt a lower priority task and force the immediate execution of the now runnable higher priority task in spite of the fact that the original lower priority task was executing in the kernel. Without this feature real time response of the system would be adversely affected by requiring kernel calls to be complete before a higher priority task could be dispatched. By allowing any processor to service any interrupt and dispatch any task, the average interrupt response time in a multiprocessor system is probabilistically improved.

It is important to note that there is no master/slave relationship among the processors in the system. Due to the symmetry of the hardware and the reentrancy of the operating system, each processor may independently execute any part of the operating system, and except for the few critical sections of the operating system, can do so in parallel with other processors doing the same. All processors are effectively identical, servicing tasks from a common task queue. This results in the most efficient utilization of the multiprocessor resources.

This symmetric and reentrant design is in contrast to some multiprocessor implementations in which asymmetric hardware architectures force a master/slave relationship between processors, resulting in excessive serialization, low processor utilization and increased complexity and overhead. In these cases,

tasks must be run only on the master processor when they request I/O or other operating system services. Similarly, non-reentrant operating systems, such as UNIX* systems as distributed by most all other vendors, are unsuitable for multiprocessor systems due to the serialization required to call into the kernel.

IV. Multiprocessing from the User's Perspective

From the user's point of view, adding processors to a HP 9000 Series 500 system simply adds resources with which to service the set of tasks that exist in the system at any given time. No changes are required to the user's application program or to the installed operating system. Tasks are not aware of the number of the number of processors running in the system. With this model, any multitasking operating system could benefit from better service through multiprocessing. The tasks in the system, corresponding to processes in an HP-UX system or partitions in a BASIC system, are serviced by the available processors in much the same way as passengers are serviced by ticket agents at an airport. Adding a processor is analogous to opening another ticket agent window.

The HP-UX operating system naturally encourages the proliferation of many independent processes. This typically leads to a relatively large population of tasks which compete for the services of a smaller number of processors. When the ratio of runnable tasks to available processors is greater than one, the operating system selects the N tasks to be executed on the N processors based on priority. In general, this leads to efficient utilization of the multiple processors.

Single programs, however, do not naturally benefit from multiple processors since a single task can execute on only one processor at a time. Single applications can benefit from multiple processors if the programs are decomposed into multiple cooperating processes. In order for a single program to be implemented as a group of independent tasks, it is usually necessary for the tasks to communicate and synchronize. The HP-UX operating system provides a wealth of facilities to aid in this effort, including shared memory, inter-process messages, a general semaphore mechanism, shared files, and pipes. The BASIC language system, which implements multitasking through the concept of an independent BASIC partition, provides shared files and memory resident volumes for the communication and sharing of data, and events for the synchronization of partitions. The usefulness of separating an algorithm or program into separate tasks is dependent on the overhead involved in synchronizing and sharing data, relative to the speedup obtained through parallel execution.

V. Performance of Multiple Processors

The performance of a multiple processor system and the incremental performance gained by adding an additional processor is dependent on the characteristics of the set of tasks being executed by the system. Different tasks demand different resources from the system and in varying amounts.

Assuming that sufficient memory exists in the system to satisfy the virtual memory requirements of all tasks in a system mix, the primary factor in determining the incremental performance of additional processors is the memory bandwidth requirements of the tasks in the mix. Tasks require memory bandwidth both for instruction fetching and for data access. Some instructions are longer running

than others, resulting in some variation in the instruction bandwidth requirements of different tasks. The data bandwidth requirements of different tasks vary over quite a large range. At one extreme are memory intensive tasks that access memory very quickly using microcoded block move or scan instructions. At the other extreme are floating point intensive tasks, which when running on a processor with a microcoded floating point implementation, use very little data bandwidth and instead spend most of their time executing microcode with very little main memory demand.

The table below shows the incremental performance obtained by the addition of processors in systems executing several different workloads. All performance figures are normalized, where unity is the performance of a single processor system executing the given workload. Any fraction greater than one indicates the incremental performance obtained by the additional processors relative to the performance of a single processor of the specified type.

Performance of Multiple Processor Configurations
Relative to a Single Processor System
for Various System Workloads

System Workload	1 CPU	2 CPUs	3 CPUs
Floating Point Intensive with Microcoded Floating Pt.	1.0	1.98	2.92
Floating Point Intensive with Hardware Floating Pt.	1.0	1.92	2.64
Integer Math Intensive	1.0	1.87	2.50
16-user Computational Workload	1.0	1.75	2.23
16-user Compilation Workload	1.0	1.65	2.05
Block-Move Intensive	1.0	1.40	1.65

The incremental performance gain of additional processors is diminished when Memory Processor Bus contention results in decreased memory bandwidth to the individual processors. In the above performance measurements, system loads with the least memory bandwidth requirements benefited most from the incremental processors. Processors with microcoded floating point show the largest incremental performance gains as processors of the same type are added in a highly floating point intensive workload. The incremental performance relative to the first CPU is not as great when processors with hardware floating point are added to other processors of the same type, although the absolute performance in all cases exceeds that of the microcoded floating point processors in the same configuration. Since the hardware floating point instructions are several times faster than the microcoded versions, the programs execute faster and demand a higher memory bandwidth. This decreases the incremental effect of additional

processors while providing higher absolute performance. For memory bandwidth intensive system loads such as those frequently executing the microcoded block-move instructions, the Memory Processor Bus becomes the limiting factor to performance. When a larger number of users share the system, other complex interactions involving virtual memory and disc access can affect multiprocessor performance, as in the 16-user workloads above.

VI. System Hardware Configuration and Performance

The multiprocessor performance data presented in the previous section represents the case when the code and data of all programs are resident in the main memory of the system. The virtual memory feature of the HP-UX operating system introduces a new aspect to multiprocessor system configuration. Virtual memory allows programs or sets of programs to execute in the system in spite of the fact that they may not totally fit in the system's main memory at the same time. The HP-UX operating system's virtual memory manager transfers parts of the code and data of the executing programs between main memory and secondary disc storage as demanded by the system. When an application program references a memory object that is not resident in the main memory of the machine, the task is blocked until the system makes the memory object resident by swapping it in from the disc. This operation is transparent to the application program, except for the long delay experienced by the memory reference.

The amount of swapping of memory objects that occurs between main memory and the swapping disc is dependent on the amount of main memory in the system and the memory demands of the tasks executing in the system. If the system is able to allocate main memory to satisfy all the memory demands of the executing tasks, then no virtual memory swapping is needed and all memory references are fast. If the memory demands of the executing processes exceed the capacity of the system's main memory, then some memory references will appear to be several orders of magnitude slower due to the need to swap the memory object in from disc. The response time experienced by any one application program can degrade non-linearly as more programs are added to the system due to the need to rely on much slower secondary memory. In general, the addition of main memory will decrease the virtual memory swapping activity for a given system workload and result in improved average memory access time.

This virtual memory phenomenon complicates the performance analysis of a multiprocessor system. A well functioning system requires a proper balance of processor, memory, and I/O resources that is well matched to the system workload. If the memory demands of the system workload far exceed the capacity of main memory, additional processors are unlikely to result in significantly increased performance, since main memory and virtual memory swapping will present a bottleneck. Similarly, if the system workload is I/O-bound, and most of the processes are typically waiting on I/O, then additional processors will not improve performance. Because of the fixed number of slots (12) in the Memory Processor Module, tradeoffs between additional processing power and memory and I/O resources must often be made. Higher density memory boards, such as the one megabyte memory card, increases the system configuration options by providing a large amount of memory at the cost of only one slot.

A HP-UX system monitoring tool allows a system engineer to dynamically view many characteristics of a system executing a given workload. This information can

be used to determine if the performance of a system workload can be improved by the addition of processors and/or memory. The monitor dynamically displays the utilization of the processors and discs in the system, the amount of virtual memory traffic, the frequency of virtual memory faults, the allocation of main memory, and the processor and memory characteristics of individual processes. It can also provide some subjective judgements of the state of the system and suggests steps that could be done to improve system performance.

Adding additional processors or memory is a trivial operation of shutting down the system, sliding the additional processor or memory boards into the Memory Processor Module, and re-booting the system. No modifications are required to the hardware boards, to the operating system boot area, to the existing memory or I/O resources, or to the application programs.

VII. Conclusions

For systems seeking additional performance, multiple processors provide a smooth growth path that is usually less expensive and less disruptive than the acquisition of a new machine. The HP 9000 Series 500 provides a very efficient multiprocessor implementation through the support of both hardware and software. Its symmetric architecture and reentrant operating systems yield maximum utilization of all processors and complete transparency to all programs. Significant performance gains can be obtained through multiprocessing for most system workloads. System monitoring tools can be used to determine the optimal and most cost effective configuration of memory and processor resources.

9013

HP EGS: A Facility Management Tool on the HP 9000 Series 200 and 300 Computers

Joe Eyre
and
Ed Brovet

Fort Collins Engineering Operation
Hewlett-Packard Company
3404 East Harmony Road
Ft. Collins, CO 80525

Abstract

This paper describes the use of the Hewlett-Packard Engineering Graphics System (HP EGS) for facility management and engineering. Primarily a 2-dimensional computer graphics system designed for engineering applications, HP EGS has many features specifically useful for facility engineering. Some of these features include its file system, ease of customizing, and outputs that enable post-processing of drawing data for additional information. A section of this paper also describes the advantages of HP EGS over other facility management systems.

Introduction

If you have ever been a member of a growing department, you can understand the following scene: Your department is growing both with people and equipment. Up until now you've managed to shuffle desks and pack your people in a little tighter. Now, however, you cannot see anyway to squeeze additional space out of the area.

You've considered your options – make the best of the area you are in, move your department to a larger area in the facility or to another facility, or construct a new building. The options dwindle when you consider that the employees are not anxious to move and the cost of a new facility is out of the question. Looking around, you search your area for additional space, not sure if it is really being used efficiently.

The manager of this department must consider more than just floor area. The department, for example, might be charged a "rent" of $1.50/square foot/month for the area it occupies in the facility. By occupying only a 100-by-100 foot area, its rent to the facility is $15,000 per month. Multiply this figure by 12 months and the charge becomes significant. And for a company that has several large facilities, each occupying around three-quarters of a million square feet, a significant amount of assets is tied up in real estate.

Facility Engineering: Managing Floor Space and Utilities

Facility Engineering is the planned placement of equipment and utilities in a facility so as to promote an effective and efficient work environment. Although manufacturing areas differ from office space in appearance, both require a layout that promotes efficient processes and communication. Facility Engineers must also modify or redraw architectural drawings to reflect the actual construction of a new addition or facility.

Most large companies have developed a Facility Engineering group to manage and coordinate the use of floor space and supporting utilities. Smaller companies and departments within companies often "volunteer" someone to design and sketch the furniture and equipment layout. In either case, designers must create or revise the floor and utility drawings each time the department adds furniture or equipment. Accurate, up-to-date, and clean drawings are essential for efficient planning of an area. They are also necessary for job bidding by outside contractors.

One of the most expensive aspects of any area change is the utilities. To add a new piece of equipment, service personnel may have to remove, add, or modify telephone, plumbing, and air lines, sprinkler systems, HVAC ducting, or computer networking cables. Drawings showing only the pertinent utilities and landmarks (such as columns) enable the service personnel to more easily make the changes. In addition, up-to-date drawings prevent unexpected discoveries such as finding that a utility is already at capacity or worse yet, no longer exists in the area.

Finally, every company requires space and equipment accounting. The Facility Engineering group not only has the responsibility to maintain the site drawings, but also regularly reports the amount of floor area occupied by each department. Because the designers have to calculate the floor area from drawings, irregular floor arrangements require a fair amount of estimating.

Facility Engineering with HP EGS

Many companies have taken HP EGS from their engineering labs to their Facility Engineering group. Although HP EGS is a low-cost, 2-dimensional drafting system, it features many of the same capabilities as facility management systems costing two to eight times as much. Facility Engineers have found that their group's productivity increases by 30 to 50% over manual methods. This increase in productivity extends beyond the drafting board; it also includes the time savings by everyone using HP EGS drawings.

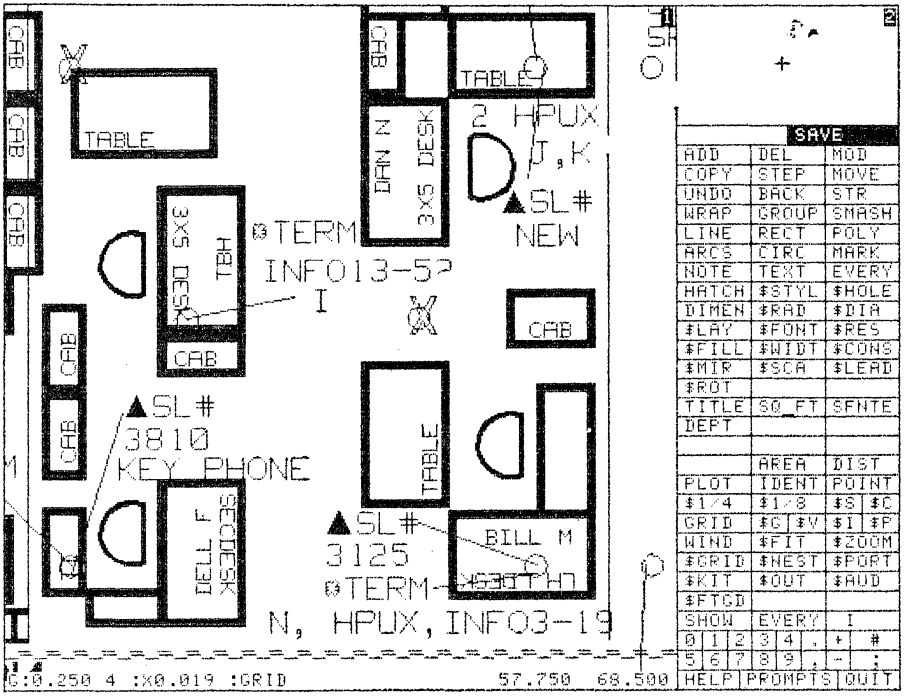
Because it operates on Series 200 and 300 computers, HP EGS can operate in a stand-alone configuration as well as in a networking system. In addition to file sharing, networking also allows designers to share storage discs and peripherals.

HP EGS with the Series 300 Computer

The following sections describe some of the benefits of using HP EGS for Facility Engineering.

Ease of Use

Designers access HP EGS either by selecting items from a menu or by entering HP EGS commands. An item can be selected from the menu by a graphics tablet or mouse, depending on the type of computer used. A representative screen display appears below. The menu is on the right side of the screen.



A Facility Drawing on HP EGS

Because of the consistent and intuitive menu structure, and the documentation included with the system, designers and engineers soon become productive with HP EGS. A tutorial quickly brings them up the learning curve by providing instructions and examples on how to use the menus and other HP EGS features.

Sharing Drawing Data

In the simplest sense, designers can share HP EGS drawing data by sharing drawing files contained on flexible discs or a Shared Resource Manager (SRM). On a larger scale, Facility Engineering groups often develop unique HP EGS personalities for their convenience. Regardless of the differences between HP EGS personalities, drawing files can still be transferred. The ability to share drawing data between groups as well as among groups encourages common standards and library parts – benefits anyone using the drawings can appreciate.

The HP EGS Archive file enables designers to use data with HP EGS post-processors and user-written post-processors. One HP EGS post-processor is the bi-directional IGES Translator. It converts HP EGS drawing data into the Initial Graphics Exchange Specification format for

transfer to other CAD systems that support IGES. Because it is bi-directional, it converts IGES drawings from other systems to HP EGS drawings.

Keeping Historical Data

By not altering old drawings each time an area change is made, HP EGS allows a department to keep excellent historical data. In contrast, paper or mylar drawings must either be erased, thus losing the previous record, or attached as an addendum sheet to the new drawing. In both cases, the manually-revised copy is not as easy to understand as HP EGS drawing files. Also, the storage space required for HP EGS drawing files is significantly less than the floor space required for drawer files that contain full-sized paper or mylar drawings.

Using Library Parts

A library of furniture, equipment, and symbol parts has several advantages. One of these is consistency; all designers will use the same parts in every drawing. Stemming from consistency is accuracy because with the same parts, different drawings will not contain parts drawn at slightly different scales.

HP EGS only "references" library parts rather than actually containing parts on a drawing. This means that the drawing file does not contain the part but only a pointer telling HP EGS to go and get the part from the library each time the drawing is retrieved. This method of pointing to parts rather than containing them is beneficial because if you must change a part, you only need to change the library part once. Then each time you retrieve a drawing that contains the modified part, the change is automatically made.

The most significant advantage of library parts, however, is speed. Once a designer creates a library part, he can add it to an HP EGS drawing with a command or macro. This is much faster than drawing the part from scratch or by using a template. In addition, a part can be duplicated and placed anywhere in the drawing with a single COPY command.

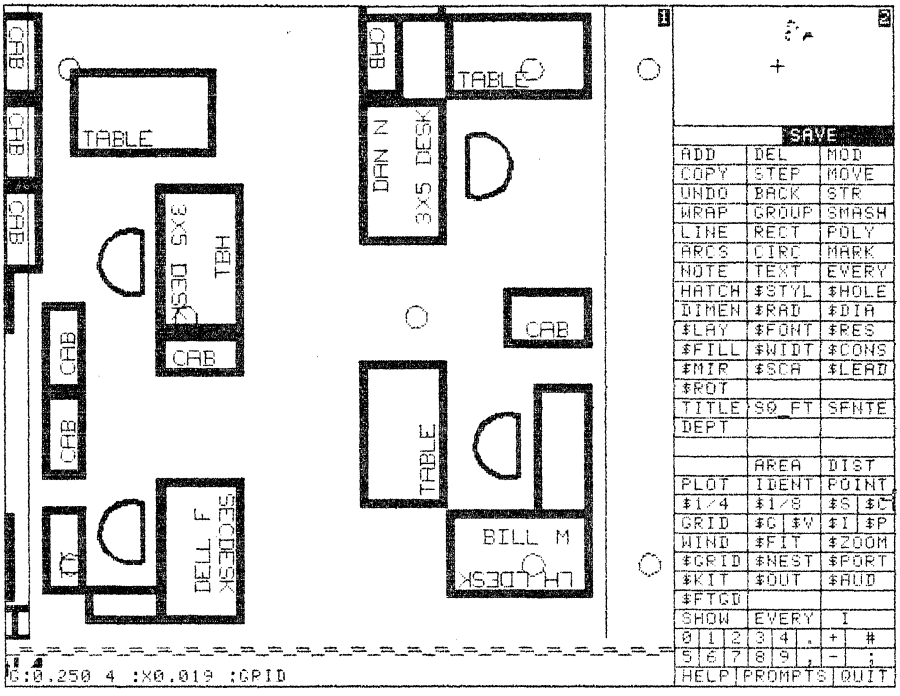
Making Accurate Drawings

With up to 100 million system points per user unit (typically a foot), HP EGS is capable of producing accurate drawings. Although Facility Engineers may not need that degree of accuracy, they can use the GRID and UNITS commands for precise layout. The GRID command determines the "fineness" of the grid (a fine grid allows accurate parts placement, a necessity for placing connecting modular furniture). Designers use the UNITS command to set drawing units (such as feet or inches).

Two additional aspects of Facility Engineering require accurate drawings. One aspect is estimates by outside contractors. With accurate drawings, they can give more realistic bids. And finally, department billing for floor space depends on accurate drawings.

Overlaying Drawings

Facility Engineering groups often take a skeleton floor layout and place on it one utility or aspect of a floor layout. For example, a drawing might contain only the furniture and any pertinent landmarks (such as columns). Using this drawing, a mover can place the furniture without being distracted by irrelevant details.



The Furniture Layer

With HP EGS, a designer can separate a drawing into as many as 255 layers. Layers are distinguished on the screen by colors; each layer is assigned a color that can easily be changed temporarily or permanently. A designer can view an individual layer of a drawing or overlay layers on top of each other. For example, where does one utility cross another? Similarly, a plotted drawing can contain from one to all 255 layers. When plotted on a clear material such as mylar, HP EGS drawings overlay just as the layers on the screen.

Post-Processing HP EGS Drawing Data

HP EGS offers several post-processors to extract additional information from its drawing data. Two HP EGS post-processors used by Facility Engineers include the HP EGS Material Lister and HP EGS Connection Lister. The HP EGS Connection Lister details connectivity information. Taken in the context of Facility Engineering, a designer can use the Connection Lister to see who is connected to a telephone or computer networking cable.

The HP EGS Material Lister, in contrast, counts parts. Take this further and you can use it to count types of equipment or furniture. If necessary, you can break "type" into finer categories such as model, date of purchase, value, function, or size. By running the Material Lister on the drawing, you can have a detailed accounting of equipment in minutes. In addition, the Material Lister produces an input file that can be used to annotate the drawing with a bill of materials.

Because of the open data structure of HP EGS, engineers and designers can easily write post-processors in BASIC or Pascal. A post-processor could be used, for example, to find the length of wire in a circuit. Similarly, if the width of a line designated the area of a pipe, a post-processor could find the capacity of the pipe by multiplying the line width and length.

Customizing HP EGS for Facility Engineering

Unlike many other facility management systems, HP EGS is very customizable. Customizing means convenience; designers have modified everything from the way the menu appears on the screen to the library parts and macros contained on it. Customizing of an HP EGS personality primarily involves modifying drawing settings and the screen menu, creating parts and macros, and adding any additional features.

A designer can modify the screen menu to contain commands and parts in addition to, or in place of, those included with an HP EGS personality. The graphics tablet is also available for additional command and part input. Some items designers have placed on menus include partitions, electrical and mechanical symbols, furniture, terminals, and commands to retrieve a drawing of a particular area in the facility.

Macros are a time- and effort-saving feature. Macros are hierarchical; they build upon previous commands and/or macros. A designer might write a macro, for example, to retrieve a department drawing that shows only the tombstones on the floor. Another macro might calculate the area between four of the tombstones. Both of these macros could then be placed on the screen menu for one-touch activation.

HP EGS also has "macro instances" for creating customized library parts. Similar to a macro in that they accomplish several tasks with one command, macro instances also enable you to add differently-sized components (such as desks) when you need them. Because macro instances enable you to create a "family" of parts from one library part, they simplify your parts library and save disc storage space.

Planning Ahead with HP EGS

HP EGS encourages effective facility planning because it is an easy and convenient tool for making "what-if" layouts. With polar or rectangular step and repeats, a designer can quickly lay out a department. This is obviously faster and more efficient than cutting out little pasteboard shapes of desks and file cabinets and placing them on a floor plan or worse yet, arranging furniture by pushing it around until it fits.

But HP EGS can be used for more than "eyeball" planning. For example, an engineer could check the physical capacity of a bus carrying telephone wires by designating its capacity in terms of width. Each telephone line also has a width. Every time a telephone line is strung through, the bus fills a little more. When the bus line is a solid with telephone lines, the engineer knows the bus is filled. This visual planning prevents time lost by the service person who discovers that the bus is already full.

Post-processors can be used for planning. Consider the previous example, that of a bus carrying telephone wires. The available circuit capacity of the telephone line can be estimated by combining a couple of post-processors. Use the HP EGS Connection Lister to find the number of occupied outlets on the line. Then operate a post-processor that counts the length and width of wire (representing the gauge of the wire) and combines this information with the number of occupied outlets. With this information the post-processor can give a usage figure that can be compared to the capacity of the line. This same idea could apply to plumbing, air lines, sprinkler systems, HVAC ducting, and computer networking cables.

The idea of post-processing line widths and lengths has another use by Facility Engineering. Think about the number of times departments communicate within an organization. Translate the number of interactions, say per week, into line widths. For example, if two departments communicate with each other twice a week, denote this with a line width of two. Connect all the departments in an area with lines representing the number of times they communicate. You can then find, either visually or with a post-processor, the most efficient arrangement of departments by minimizing the length of "fat" lines – those lines representing frequent communication.

And finally, you could use a post-processor to count all the floor space not occupied by furniture or equipment. By studying the amount and location of this floor space you could identify wasted or inefficiently used space.

The HP EGS Advantage

HP EGS has several advantages over other computer-aided facility planning systems. To many companies, the most significant advantage is cost. A single HP EGS system costs one-half to one-eighth that of many of the full-featured facility planning systems available today. Most Facility Engineering groups do not need many of the computational and long-term planning features offered by these systems. Multiple HP EGS systems not only decrease in price per system, but also cut down cost by allowing several designers to share plotters and other peripherals.

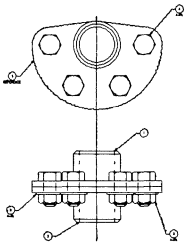
Unlike many of the more expensive facility planning packages, HP EGS is easily customizable. Starting with one of the HP EGS personalities, you can develop a personality and parts library specifically tailored for your needs. And even if you have a customized personality, you can still pass HP EGS drawing data to other designers, regardless of how they have modified their HP EGS system. HP EGS data is easily transferred to other workstations via an SRM or flexible discs and to other CAD systems via the IGES Translator. Post-processors can access the data via the Archive file.

HP EGS operates on Series 200 and 300 computers allowing other operating systems and applications software to be run alongside HP EGS. One especially useful application is HP TechWriter, a document editor that enables you to merge HP EGS graphics with text. HP TechWriter is useful for office documents such as proposals on how to develop an area in the facility. You can then print the document complete with pictures on a dot-matrix printer or on the HP LaserJet. An example of HP TechWriter output appears below.

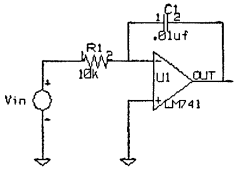
HP TechWriter

HP TechWriter is an engineering documentation product that provides Series 200 and 300 customers with a version of the familiar Pascal text editor that has been enhanced to allow customers to create and print illustrated documents. The illustrations are contained in plot files produced by graphics editors such as HP EGS or by customer programs.

HP EGS drawing:



Pictures are drawn scaled to a size specified by the picture command. If you do not need to see the pictures, HP TechWriter allows you to turn off the display of your figures if you wish to. The editor also provides a number of convenient features for developing documents, such as right justification, multi-line headers and footers, and automatic table of contents generation.



Side by side text and graphics allows you to produce annotated pictures on any printer which can support this feature. Local margin commands allow you to use special margins for quotation text or the text describing your picture, or to turn margins off completely for tables.

This sample was produced on an HP LaserJet printer (HP 2686A) showing its variable resolution graphics capability - the top picture is at 150 dots per inch while the lower picture is at 75 dots per inch.

Sample of HP TechWriter Text with HP EGS Graphics

Behind every HP EGS system is a trained HP EGS Product Specialist who is readily available for information and consultation. Many HP EGS users groups exist and more are forming as the 2500 system installed base grows.

Throughout this paper you have seen the benefits of using HP EGS for facility engineering and how to get more from the system by post-processing data and customizing. This final section listed some of the advantages of HP EGS over other facility planning systems.

Acknowledgements

Many of the ideas for this paper were suggested by the Facility Planning group at the Hewlett-Packard Fort Collins facility. Thanks are also due to Tom Krantz for his enthusiasm and creativity with HP EGS, and to Gretchen Tobin, Mary Ann Moore, and Ron Mora.

9014. HP9000 JOB ACCOUNTING - HOW TO FIND OUT WHAT YOUR HP9000 IS DOING

Harold Parnigoni
Northern Telecom Ltd.
150 Montreal-Toronto Blvd.
Lachine Quebec
Canada H85 1B6

ABSTRACT

This paper will detail the log files available on the HP9000 model 540. Also, shell scripts and data collection programs that can assist the system administrator to find resource bottlenecks and report system utilization will be shown. This paper basically will show how to report to management the utilization of your HP9000 and allow one to have documentation to backup equipment purchase requests.

9015

HP TechWriter: Integrated Text & Graphics for the HP9000 Series 200

Roy E. Anderson
Member Technical Staff

Fort Collins Engineering Operation
Hewlett-Packard Company
3404 East Harmony Road
Fort Collins, CO 80525

ABSTRACT

This paper describes HP TechWriter, an integrated text and graphics product aimed at the documentation needs of engineers and scientists. HP TechWriter provides text and graphic integration in an interactive editing style that approaches "what-you-see-is-what-you-get," runs on all of the HP9000 Series 200 computers, and supports a range of printer technologies.

In addition to a feature overview, this paper provides descriptions of the key features and how they are accomplished, and presents execution times for some typical operations.

[Note: This paper was written with HP TechWriter. The camera-ready copy was produced with a LaserJet printer.]

1. INTRODUCTION

HP TechWriter is designed to aid scientists and engineers in doing their job of working with words and ideas in a more effective manner. In addition to typical word processing capabilities, the documentation of technical ideas often requires the definition and clarification that can only be achieved through graphical means. Technical users need a product that allows them to merge standard word processing capabilities with graphical illustrations. Furthermore, they need a product that is responsive, easy to use, and available on

a personal workstation.

HP TechWriter provides a document development environment in which a document can be easily evolved from initial concept to finished product. Throughout the document's stages of development it always appears on the computer screen in the same format that it would if it were printed, thus eliminating the need for frequent printings during the preliminary writing stages. A few exceptions to an exact "what-you-see-is-what-you-get" philosophy exist for increased performance and ease of use. These exceptions are page breaks and pagination related features such as headers and footers. Paragraph margination and other formatting features occur real-time and with considerable speed. Pictures are displayed on the CRT in proper relation to their surrounding text, as they will appear on the printed page. Text and pictures can be scrolled up or down in a continuous fashion at a comfortable speed.

Section two describes in more detail the key features and design goals that distinguish HP TechWriter from other technical documentation products, and section three describes in a general manner the implementation techniques that were used to achieve them. Finally, section four provides sample performance data on several of the types of operations and functions described in the paper.

2. FEATURES

HP TechWriter offers the typical text creation and modification features found on most interactive text editors. Principle among these are:

- continuous scroll
- find/replace
- settable tabs
- arbitrary movement to any location
- arbitrary insert/delete
- buffered insert/delete for easy duplication
- copy/move arbitrary text blocks
- whole or partial disc file insertions

In addition there are also features specifically aimed at document preparation, such as:

- left and right justification
- ragged left or right margin
- top and bottom margin control
- auto-indent/outdent
- centering
- multiple line spacing (any value)
- optional automatic word wrap

- multi-line headers and/or footers
- flexible page number output
- conditional and unconditional page break control
- selectable non-print portions of document
- print-time character substitution
- table of contents
- underline
- over-strike

2.1 User Interface

Performance is a primary design goal of HP TechWriter - it is difficult for an accomplished typist to get ahead of the program, so characters normally appear on the CRT the instant they are struck on the keyboard. Operations like copying or deleting information, jumping to a new location, search and replace, and margination are tuned to minimize the disruption from the user's point of view.

The user's interface is based on single character commands - causing it to be slightly more difficult to initially learn than softkey or iconic based interfaces, but with the distinct advantage that it becomes nearly transparent with a small amount of practice (most users claim this happens in one to two weeks of use). Thus, the mechanics of operating the editor require almost no conscious thought, freeing the user's mind to concentrate on "what to write," not "how to write it."

There are only thirteen *primary commands* in the editor for use in creating and modifying text. In addition there are nine *document commands*, which are textual information entered into the body of the document to control formatting, pagination, and graphics integration features.

2.2 Graphics Integration

HP TechWriter is **not** a graphics editor product; that is, it cannot be used to create a picture. Instead, it captures graphics data stored as a *plot file* on disc by another program, and integrates it into a document. The user can specify any position relative to the document's text, and the size of a rectangular area to be occupied by the picture in terms of character line and column positions. HP TechWriter then scales the picture to fit in the described area whenever it is drawn. The rectangular area for any picture can range from a single character cell to a full page.

HP TechWriter integrates pictures from graphics software that can produce ASCII files containing Hewlett-Packard Graphics Language (HPGL) plotter commands. Some of the products that

have this capability include:

- FARB Computer-Aided Design System
- SMT Inc.'s Equation Writer
- HP Engineering Graphics System (HP EGS)
- HP Data Grapher/200
- HP Graphics Editor/200
- HP Statistics Library/200
- HP Project Management/200
- HP Graphics Presentations/200

Perhaps even more important is the flexibility that permits user written programs (in BASIC or PASCAL) to produce graphics output that can be subsequently included in HP TechWriter documents. Programs that provide graphical information unique to a customer's specialized applications can now be included in written reports.

For some limited types of monochromatic graphics applications where resolutions beyond 300 dots per inch are not required, HP TechWriter can reduce, or eliminate, the need for a plotter by effectively utilizing a graphics printer instead. Typically, printing is as fast, or faster (depending on resolution), and more convenient than plotting for small pictures.

2.3 Multiple Printer Support

A design objective for HP TechWriter was to enhance the value and useful lifetime of existing HP dot matrix printers that had been sold for several years prior to the introduction of HP TechWriter, as well as to more effectively utilize the new printers and printer technologies being introduced in the HP TechWriter time frame. We felt it was a significant contribution to our users to offer a documentation product for the scientific marketplace that would operate with a price and technology range of printer products, instead of just one or two very capable (and possibly more expensive) printers. HP TechWriter currently supports eleven printers ranging from the HP 2631G and HP 9876A to the ThinkJet and LaserJet. In this case, "support" not only means that a particular printer has the character set and control sequences expected by HP TechWriter, but it also means that the program contains the printer's attributes needed to properly align and integrate text and graphics on the printed page.

Rather than tune HP TechWriter's features for a specific printer, we adopted a generalized approach so that the features could be accomplished by most, or all, of the supported printers. For example, instead of offering bold face that only some of the newer printers could do with an actual font, we supplied an over-strike capability that all of the impact

printers can do to simulate a bold face effect. In addition, we included the general capability to send any user-defined escape sequence to the printer to allow the user to select any font feature that a particular printer may have (now or in the future).

In the same vein, we chose not to provide a *true* multi-font capability to support mathematical formatting because it could not be supported by most of today's printers. Instead, we chose the more general solution of graphics integration which enables practically unlimited freedom of font expression and formula layout, all within the capabilities of our lowest priced printers. An example of the result of this philosophy is the *picture* of an equation shown below. This picture was produced with SMT's Equation Writer, a graphics program specifically designed for producing mathematical equations as pictures, potentially for inclusion in HP TechWriter documents.

$$f(x,y) = y + \frac{\int \frac{y + \frac{y+1}{x}}{y+4} dx}{y+1}$$

3. IMPLEMENTATION TECHNIQUES

3.1 Data Structures

In order to achieve the highest performance possible with the Series 200 processors, HP TechWriter employs a very simple data structure for the representation of a document. This approach lends credence to the popular belief that "simpler is faster" in computer science. The representation of a document is essentially the same on disc, where it is stored permanently, and in memory where it is stored in its entirety during an editing session. This reduces storage and retrieval operations to linear copies done at the beginning and end of an editing session, and enables them to be accomplished at the transfer rate of the mass storage device or its interface.

The document's data structure is the ASCII text itself, stored contiguously with a single 1024-byte record at the beginning that contains environmental parameters (e.g., margin and tab settings, location markers, and search/replace operators). The concept of *document commands* simply represented as ASCII text embedded in the document permit inherently fast editor manipulations, as well as allowing easy transfer of data to other computer systems.

Editor operations like insert and delete, which require linear copies of portions of the document because of this simple data

structure, occur with surprising speed. This speed is due to the fact that nearly all of the Motorola MC68000's processor cycles are devoted to the copy operation, which is taking place in memory with effective use of specialized copy instructions.

3.2 Graphics Integration

The integration of text and graphics is accomplished only by textual reference (via a *document command*) within the data structure and by merging the two forms of data on the CRT or printer at display time. This allows HP TechWriter to take advantage of raster displays and printers that have built-in alpha raster capabilities. In addition HP TechWriter can send text data to them in the normal ASCII encoded manner, achieving a throughput rate that cannot be attained by today's graphic-only (bitmapped) devices.

Figures are stored on disc by the graphics programs that generate them, in the form of ASCII plotter instructions using HPGL as a format standard. During an editing session, HP TechWriter performs the vector-to-raster conversion by reading the identified figure's file and "plotting" it to the CRT display memory, whenever the figure comes into view on the CRT, or is encountered during a listing of the document to a printer. This solution avoids having to reduce the memory space available for a document's textual data in order to provide space for graphics data, or limiting the number of figures permitted in a document, or a combination of both. HP TechWriter documents can reach the same maximum textual size (as constrained by the quantity of computer memory) with or without references to graphics figures, and the number and size of figures referenced is virtually unlimited.

3.3 Raster Graphics Printing

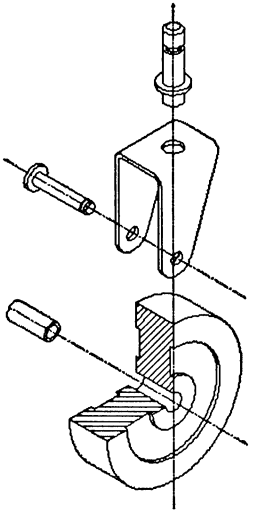
Figures are output to a printer by two distinct mechanisms depending on which HP TechWriter program is being utilized. The editor, as described above, "plots" a figure in the CRT memory using the CRT's size and resolution as upper limits for the figure's representation. The pixels of the plotted figure are then sent to the printer a row of pixels at a time. This technique, while making good use of a "free" portion of memory (i.e., the CRT display memory), suffers two shortcomings:

(1) the size of the figure on the printed page is limited by the number of pixels in the horizontal and vertical direction on the CRT (i.e., its resolution); and,

(2) it is usually impossible to obtain the printed alignment of figures and adjacent text as it was shown on the CRT.

The second shortcoming is a result of differences between the amount of graphical space occupied by an alpha character on most combinations of CRT's and printers. When there are more graphic pixels per alpha character on the CRT than the printer, the printed picture appears to "grow" relative to adjacent text; when there are less pixels per alpha character on the CRT than the printer, the printed picture appears to "shrink" relative adjacent text. The editor balances the excess size of the figure, or white space around the figure, but nonetheless produces only *draft quality* output as a result.

The second mechanism for figure output, employed by HP TechWriter's lister program, was designed to address the editor's shortcomings. This program is a "post-processor" of from one to fifty files created by the editor. Performance is still a primary goal. However, since there is no interactive editing requirement, the lister can process documents sequentially by reading a line at a time from disc, thus its data memory requirements are minimal compared to the editor's. The lister uses this extra supply of memory for the vector-to-raster conversion of figures at any resolution required by the designated printer in order to obtain a match between the appearance of the figure relative to its surrounding text when it was displayed on the CRT and on the printed page. A Series 200 computer with 512K bytes of memory is sufficient to produce pictures as large as 8 by 10 inches at 100 dots per inch. As a result, the lister produces *final quality* document output.



With both mechanisms, text can be positioned beside or "on top of" a picture by sending a row of alpha characters followed by a carriage return (but no line feed) and then the number of pixel rows appropriate for the height of the printer's character set. Ten of the eleven graphics printers supported by HP TechWriter can produce text and graphics merged in this fashion. This paragraph and the picture to the left (drawn with HP EGS and printed here with the lister at 150 dots per inch), are an example of this capability.

4. PERFORMANCE EXAMPLES

4.1 Data Memory Size

This paper, in its final form contained 21,500 bytes of data, about 25% of the maximum size file that could be edited on a 512K-byte Series 200. Since there is no additional overhead for larger amounts of memory, machines with more than 512K-bytes devote all additional memory to document data space. At an average of 2500 bytes per page, a 512K-byte machine has room for about 40 pages. To overcome this limitation, the HP TechWriter lister enables up to fifty files (about 2000 pages using these averages on a 512K-byte machine) to be output as a single document, or a series of separate documents.

This paper contains 450 lines of text information and an additional 23 lines of document commands - in this case, a 5% overhead for formatting data.

4.2 Real-time Margination

To change this paragraph from the margin settings used in the rest of the paper (64 columns wide) to twenty columns wide, as shown below in italics, took 280 milliseconds with a justified right column. With a ragged right column it took 124 milliseconds. To change the entire document to twenty-column-wide margins (there are 41 paragraphs) took nine seconds with, and four seconds without, right justification.

To change this paragraph from the margin settings used in the rest of the paper (64 columns wide) to twenty columns wide, as shown below in italics, took 280 milliseconds with a justified right column. With a ragged right column it took 124 milliseconds. To change the entire document to twenty-column-wide margins (there are 41 paragraphs) took nine seconds with, and four seconds

without, right justification.

4.3 Data Insertion and Deletion

To insert the narrow paragraph shown above (24 lines) at the front of this paper using a copy command (this is an operation equivalent to "insert" but involving no typing time) took 420 milliseconds. To delete it took 310 milliseconds. To do the same operations at the end of the document took 270 and 190 milliseconds, respectively. The differences, 150 milliseconds for the insert and 120 milliseconds for the delete, represent the time required to copy the document's data (21,500 bytes) to a different position in memory because of the linear data structure described in section 3.1.

4.4 Graphics Plot versus Print

To illustrate the comparison between outputting a picture to a plotter and a printer, the figure shown in section 3.3 was plotted by sending the plot file produced for HP TechWriter to a 7475A plotter. The picture's drawing limits were set so that the plot was the same size as the figure in this paper prior to photo-reduction for these proceedings. It took 50 seconds to plot the picture in a single color (i.e., multiple pen selections were not a factor). It took HP TechWriter's lister program 42 and 15 seconds to print the picture on the LaserJet printer at 150 and 75 dots per inch, respectively (no text was printed with the picture for this comparison). The same print operation took 23 seconds using a ThinkJet printer at 96 dots per inch.

5. SUMMARY

HP TechWriter is a capable technical documentation product that provides fully integrated text and graphics in a convenient, easy-to-use, and responsive manner. It is supported on all the HP 9000 Series 200 computers, operates within Pascal or BASIC disc environments, and fully utilizes all the printers supported on the Series 200.

HP TechWriter was introduced on July 1, 1984, and since then hundreds have been installed. Customers are using HP TechWriter in conjunction with HP EGS to produce assembly manuals for their manufacturing operations and for internal engineering design documentation, as well as routine technical reports and memoranda. At the time of this writing there have been no defects ("bugs") formally reported to Hewlett-Packard. Of course we do not claim that HP TechWriter is an example of error-free software, but the common functions used by most people most of the time have been thoroughly tested and proven.

It is a high quality product that carries the Hewlett-Packard name with pride.

ACKNOWLEDGMENTS

Many of the ideas for HP TechWriter, especially in the area of printing text and graphics, were developed by Elaine Regelson. A special thanks to Gretchen Tobin for her frequent and helpful reviews of this paper, and to Ron Mora, Tom Krantz, and Sandi Anderson who encouraged me to write it in the first place.

REFERENCES

1. E.C. Regelson and R.E. Anderson, "HP TechWriter: Illustrated Documents for Engineers," *Hewlett-Packard Journal*, Vol. 36, No. 2, February 1985, pp.4-9.

9016. SETTING UP A DATA CENTRE COMMUNICATIONS NETWORK- A PRACTICAL EXAMPLE

Harold Parnigoni
Northern Telecom Ltd.
150 Montreal-Toronto Blvd.
Lachine Quebec
Canada H85 1B6

ABSTRACT

Many papers have discussed the benefits and pitfalls of various communications strategies. I intend to show how various different types of strategies can be put together to provide a versatile and sophisticated network, while providing high user friendliness. The use of HP3000, HP9000, IBM, PACX, leased lines, X.25, terminals, etc. in my site (NT Lachine Data Centre) will be presented as a practical example.

A detailed discussion on communications strategies for the HP9000 will also be presented.

9017. HUMAN INTERFACES FOR SERIES 200 BASIC PROGRAMS

Stephen Taylor
Hewlett Packard Co.
3404 E. Harmony Rd.
Ft. Collins CO 80525

ABSTRACT

In addition to its other strong points, Series 200 BASIC provides a number of capabilities which make it easy to implement custom user interfaces. After a general summary of past and present capabilities of the BASIC Workstation, this presentation will illustrate how these capabilities can be used and combined to provide friendly interactive interfaces, with application specific messages, menus, and even simple windows. Topics to be covered include: use of the system's input processing capabilities; menu interfaces, from softkey menus to selection lists and on to icons; and screen management techniques, including providing for both alpha and graphics windows. Attention will be given to methods of determining the resources available, and to writing the interfaces in a manner which will transport easily from one machine to another.

The presentation will conclude with a discussion of possible directions for future versions of the BASIC Workstation. A working knowledge of Series 200 BASIC will be assumed in this presentation, but on special background in human interface topics.

9018. CONTROLLING AN ULTRASONIC INSPECTION PROCESS WITH AN HP9000

G.P. "Trudy" McElrath
Martin Marietta Aerospace
Michoud Division - Department 3437
P.O. Box 29304
New Orleans, Louisiana 70189

1.0 INTRODUCTION

This paper discusses the use of an HP9000 as the central processor for an automated weld inspection system. The system described herein is currently being tested at Martin Marietta in New Orleans. It is still experimental and as such is under constant revision. Some of the ideas presented here have not yet been fully implemented. An attempt has been made to identify these issues when they are discussed.

2.0 Why the HP9000?

The HP9000 was chosen as our primary computer for a number of reasons. Our original Automated Weld Inspection System (AWIS) was purchased from a British firm called MatEval. This system ran on an HP9845B computer. While functional, it had a number of drawbacks, primarily speed and limited graphics capability. A full inspection had to be performed before data analysis could begin, and then a further wait was required while the analysis was completed before inspection results could be viewed. The methods of results display were also limited.

Utilizing the 9000's multitasking capability, we can view results while an inspection is progressing. Although still a three-step process (data collection - analysis - display), the three steps can now run concurrently. The fact that the 9000 is a 32-bit machine also speeds up processing time.

Another advantage of the HP9000 is its graphics package. The operator is now able to view weld defects in a three-dimensional representation of an area of weld. This display can be rotated as desired, enabling the operator to view the flaw from a number of different orientations.

Yet another advantage of the HP9000 for our site is the fact that it is in the same family as the HP9845B. Using the TRANSLATOR utility, programs already written for the 45 can be easily adapted for use on the 9000.

3.0 OVERVIEW

The purpose of our system is to detect flaws within a weld and inform an operator of the location and approximate size of those flaws so that repairs can be made as quickly as possible.

A remote unit, containing 20 ultrasonic transducers and a marker pen, is motor-propelled along the length of a weld. The transducers test each area of the weld as they pass it and the pen marks flaws detected. Data collected by the transducers is sent to the HP9000, where it is analyzed according to constraints the operator establishes. Analysis is done on every foot of weld

after it has been inspected. Once an area has been analyzed, the operator can display the results of that analysis in either tabular form or a graphic representation.

While an inspection is progressing, the operator may pause and/or stop it, back up the remote, and re-inspect an area of weld. The operator is also informed of any error conditions encountered. If the error is a severe one (for example, a prolonged loss of coupling in the transducer area), the system will pause the inspection and await operator instructions.

4.0 HARDWARE CONFIGURATION

The major hardware components of our system are an HP9000 model 520 computer with 512K of memory, a MatEval MicroPulse unit, a remote unit containing 20 ultrasonic transducers, and an HP9895A floppy disk drive. These are briefly explained in the following sections.

4.1 THE HP9000

The HP9000 is the operator's interface to the inspection system. Through keyboard entries, he may initiate, control, and extract results from the inspection process. The 9000 also performs all data analysis and display, and programming of the MicroPulse unit.

4.2 The MatEval MicroPulse

The MicroPulse unit, manufactured by MatEval, is a microprocessor that handles all direct communications with the remote. It runs with an Intel 8085 chip and communicates with the HP9000 through the HP-IB. By a direct link to the remote, the MicroPulse controls the transducer array, although the MicroPulse itself is controlled by commands sent to it from the HP9000. These commands contain ASCII mnemonics which program the MicroPulse to perform an inspection according to operator instructions.

4.3 The Remote Unit

The remote unit, also manufactured by MatEval, contains an acrylic block holding twenty ultrasonic transducers. These transducers are set at different angles within the block so that every section of a given area of weld is inspected. The remote also contains a marker pen which visibly marks the location of any flaws on the weld.

The remote unit is motor-propelled along the length of a weld, firing the transducers at set increments. These increments are set by the operator prior to the inspection. (A message is sent from the HP9000 to the MicroPulse, which programs the remote.) The information gathered by the remote, as well as any problems it may encounter (e.g., a motor stall), is sent through the MicroPulse to the HP9000 over the HP-IB as ASCII strings.

4.4 The HP9895A Floppy Disk Drive

The HP9895A is used for archival purposes. The raw data gathered by the remote during an inspection (and transmitted from the MicroPulse to the HP9000) is preserved for additional analysis at a later date. Although the HP9000 has an internal floppy drive for 5-1/4 " diskettes, the large amount of data gathered during an inspection warrants the use of 8" diskettes. These are handled by the HP9895A.

5.0 SOFTWARE

All of our programs are written in BASIC. They are designed to run independently of each other, interfacing primarily through a common data area.

5.1 Approximating Real-Time

In an attempt to produce real-time display of inspection results, we are running our programs in three partitions. The Primary partition handles all programs not directly analyzing or displaying inspection results. This includes the software to set up and control an inspection, and also that which handles communications with the MicroPulse. In the Analysis partition, MicroPulse messages are sorted and analyzed, whereas the Display partition displays inspection results. Communication between the partitions is handled by event flags and the Common data area.

5.2 Common Data Area

There is a COMMON area, accessible to all partitions, which contains certain variables and data structures. This common area is the primary means of communication between programs. Among things stored there are the geometric layout of transducers at the remote, the current position of the remote on the weld, a DATALOCK flag which indicates if the database is currently in use, and a NEXTFOOT location which specifies the next foot of weld to be analyzed.

5.3 System Start-up

Our system is designed to start at power-up of the HP9000. An AutoStart program will set default directory to the sub-directory containing our files and then load a main menu program. This program begins by loading the system's COMMON area.

A number of data items are loaded at this time. One of these is a table defining the layout of the transducers at the remote. For each transducer, values are listed which indicate the distance of the transducer from the edge of the remote, and also the angle at which that transducer is set. These are significant because they define the area of the transducer's beam on the weld.

Also initialized by the main program are the NEXTFOOT location (it is set to 1), the DATALOCK flag (it is cleared), and the Analysis Event Flag (it is set to 0).

The operator is then required to select (through the special function keys) one of a number of options. These options include inspection run preparation,

hardware check-out, analysis and/or display of previously collected inspection data, and display or printing of selected files that are present on the internal hard disk.

5.4 Preparing For An Inspection

During this preparation phase, the system will remind the operator of the hardware set-up required (the remote must be aligned against the weld, coupling must be activated, etc.) by displaying a series of steps and requiring the operator to confirm that they have been done. The system will then verify for itself that the MicroPulse is powered up and ready for communication by initializing it over the HP-IB. If a problem is encountered, an error message is displayed to the operator. The system repeats this check until the problem is corrected.

A significant part of preparing for an inspection is programming the MicroPulse and through it the remote. As previously mentioned, data is actually gathered at the remote site by twenty ultrasonic transducers. These transducers, or probes, are angled and timed for firing such that they run forty-six different tests on each area of the weld they pass. By programming the MicroPulse to vary a given test's gate and gain settings, the operator can manipulate which portion of the weld is checked by each test. This enables him to fine-tune his system to accommodate weld width fluctuations and other factors which may vary from site to site or weld to weld.

Another area of inspection preparation involves the analysis process. Prior to an inspection, the operator may change the criteria used in defect analysis. The new criteria will only be in effect for a single inspection run. Upon completion of that run, the values will be restored to their default values.

5.5 The Inspection Process

The inspection process revolves around communication between the HP9000 and the MicroPulse.

An inspection is initiated by operator selection of a designated special function key, which causes the runtime program to be activated and a message to be sent to the MicroPulse to begin inspection of the weld.

The runtime program contains a subroutine to handle all messages sent from the MicroPulse. This subroutine is called whenever an interrupt is received from the HP-IB port connected with the MicroPulse. This routing occurs due to an "ON INTR" statement in the early part of the runtime program.

As an area of weld is inspected, any flaws found will be flagged on the weld with the remote's marker pen and in ASCII messages sent from the MicroPulse unit to the HP9000. These messages are stored in a series of raw data files on the HP9000's internal hard disk. These files are collections of all messages sent from the MicroPulse to the HP9000, including any error conditions which may have been encountered. A given raw data file will contain information on no more than one foot of weld and will be named "FOOTx.DAT" where x indicates which foot of weld it represents. Once that one-foot area has been inspected, the file is closed and a new one is opened. The first file is then available to the Analysis routine, and the Analysis Event Flag is incremented to reflect this.

While the inspection is progressing, the operator will have the options (through special function key selection) of pausing/continuing/stopping the inspection, viewing the current analysis criteria, displaying areas of the weld that were already analyzed, and pausing the inspection to back up the remote and re-inspect an area of weld. The choices pertaining to control of the remote cause appropriate commands to be sent to the MicroPulse, which executes them. Again, these commands are ASCII strings.

Once an inspection is completed, the system will remind the operator of actions required on his part regarding the hardware (e.g., disengage coupling, remove remote from weld). The operator must confirm that these have been done. The system then provides the options of either displaying results or terminating the session.

5.6 The Analysis Process

The Analysis process is triggered by the Analysis Event Flag, which is incremented by the inspection program (running in the Primary partition) when it has closed a raw data file. The Analysis program itself is a loop beginning with a "WAIT FOR EVENT". Analysis is done on a single file each time through the loop. Since a "WAIT FOR EVENT" is used, the event flag will be decremented as a file is processed. Therefor, it will always contain the number of files ready for analysis.

Which data file is analyzed is determined by the current setting of the COMMON location NEXTFOOT. The ASCII representation of that integer is appended to the string "FOOT" and the extension ".DAT" is added. This is the name of the data file for the foot of weld to be analyzed.

Initially, our system used IMAGE and QUERY 9000 to build and access a database containing all information collected during an inspection. It was determined, however, that due to the limited access we required to the data and the large amount of memory used by the QUERY programs, it was more advantageous for our purposes to write our own data access routines.

The Analysis program processes all flaw indication messages in the raw data file. These messages indicate which tests detected a flaw and the position of the remote when the detection was made. This information is then used, along with data in COMMON specifying the beam paths of the transducers involved, to generate three location coordinates. The X-coordinate represents the distance along the axis of the weld from the beginning of the inspection, the Y-coordinate the distance from the weld centerline toward the root or crown of the weld, and the Z-coordinate the distance from the centerline toward either side of the weld. These coordinates are stored with the test number and the remote position in a temporary data file.

Once analysis on a raw file is completed, the program checks the DATALOCK flag in COMMON. If this is clear, the program sets it and then appends the temporary data file to the database. It updates the NEXTFOOT location in COMMON to reflect the foot of weld just analyzed, and then clears the DATALOCK flag. The database is now available to the Display routines.

The analysis process is normally not seen by the operator. He can, however,

through the primary partition, call up a screen which shows the analysis parameters and the location of the weld area currently being analyzed.

5.7 Displaying Results

The display program allows the user to display (in a variety of ways) any section of the weld upon which analysis has been performed. It is begun by a call from either the main menu program or the runtime program. If called from the main menu, it will present displays based on the current database. If there is no database, an error message is given to the operator and no displays are allowed.

When the operator chooses the display option (which he does from either the main menu or the runtime program), the system prompts him for display range. He must specify the start and end points (x-coordinates - distance along the weld) in millimeters. The system then checks locations in COMMON to verify that the area of weld requested has been inspected and analyzed. If analysis of that area has not completed, or if the area has not been inspected, an error message is displayed to the operator and he is requested to provide new range coordinates or abort his request.

If a legal range is specified, the program waits for the DATALOCK flag in COMMON to be clear. Once this condition exists, the display program sets the flag, copies the current database to its own data file, and then clears DATALOCK. This copy is made and used by the program so that it does not tie up the actual database and in so doing impede the Analysis process.

Display of inspection results is provided in two formats - tabular and graphic. Both formats give the operator the choice of either hardcopy or CRT screen display. Tabular format is simply a table listing what tests found flaws at which locations on the weld and the position of the remote when the flaw was seen. Graphic format provides a more diverse selection of display.

There are three drawings used in graphic display. The first is a "bird's eye view" of a typical weld, another is a side view, and the third is a three-dimensional slice of weld. These drawings are used as backgrounds when plotting weld defects. We plan to add a fourth drawing which will show the transducer layout over a weld and the beam path of each transducer.

The operator chooses which form of graphic display will be used. The program clears the screen and displays the corresponding drawing. It then marks defective areas by using the coordinates in the copy it has made of the database. Only those flaws with X-coordinates within the range specified for the display will be shown. Once the marking is completed, the system will dump the screen to the internal printer if the hardcopy flag is set. (This flag is set or cleared when hardcopy or CRT display is chosen.) The operator will then be given the options of returning to either the main menu or runtime program menu (whichever he was in when he called results display), displaying an additional area, or, if he has chosen the three-dimensional display, of rotating that display. [This rotation feature is not yet fully implemented on our system.]

6.0 CONCLUSION

As indicated early in this paper, our system is still experimental. Once our initial implementation is completed and fully tested, we intend to revise it so that it is an actual real-time system. Current plans toward this end include adding more memory to our HP9000 and a possible conversion to UNIX.