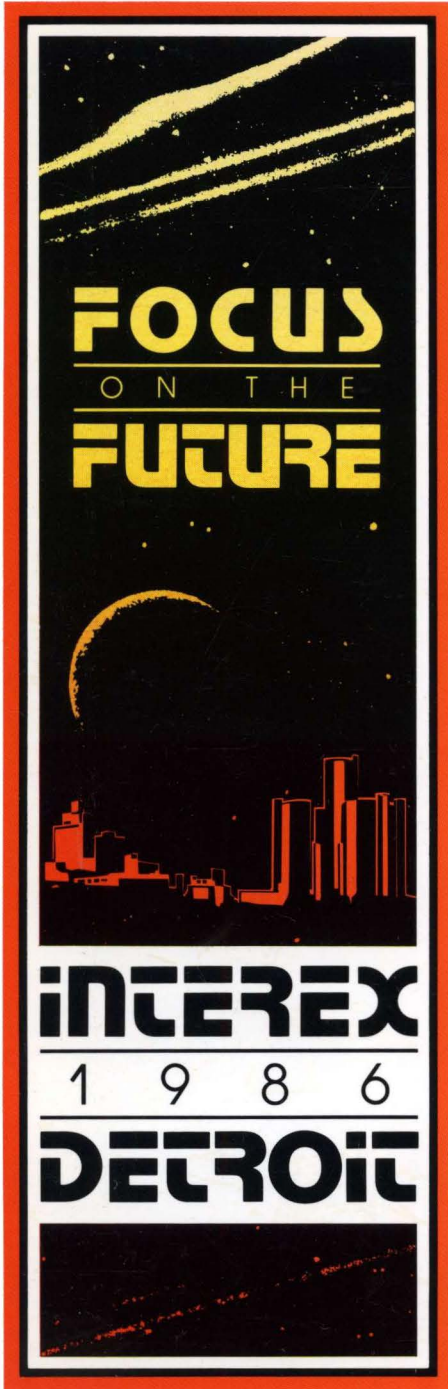


TECHNICAL PROCEEDINGS

HP 1000/9000



INTEREX
DETROIT CONFERENCE
SEPTEMBER 28 - OCTOBER 3, 1986

INTEREX

the International Association of
Hewlett-Packard Computer Users

Proceedings of the 1986 Conference at Detroit, Michigan Hosted by the Southeastern Michigan Users Group

Papers for the HP1000 and HP9000

F. Stephen Gauss, Editor

Paper Review Committee

Wayne Asp
Cimarron Boozer
John Campbell
Dean Clamons
Donald Clapp
Art Gentry
Ken Griffin
Hugh Hanks
Marc Katz
Jock McFarlane
Richard Minor
Glen Mortensen
Chris Pappagianis
Larry Rosenblum
Tim Snider
William Steele
Dan Steiger
Steven Telford
Don Wright

Introduction

This volume of the Proceedings of the INTEREX 1986 North American Conference was printed from camera-ready copy supplied by the authors. Due to the proliferation of word procesors for HP and other computers, it was deemed appropriate to request that the authors format and print their own papers, rather than submitting them in machine-readable form, as in the past. It was gratifying to the editor to find that all of the authors were able to meet this requirement, thus saving him several months of work. Papers have been numbered sequentially in order of presentation at the conference with HP1000 papers numbered 10xx and HP9000 papers numbered 90xx. Papers based on the tutorials are numbered with a T designator and appear at the beginning of this volume. Several papers will be of interest to both communities, especially as the 1000 and 9000 lines merge at the high end. It is also quite likely that papers in the companion volume for the HP3000 and Series 100 will be of interest.

Thanks go to the authors who met the submission requirements and had their papers in by the deadline. Thanks also to the paper review committee for their timely responses and helpful comments. Many of the papers from Hewlett-Packard, including a number of the tutorials, were obtained through the efforts of Pam Tower.

Finally, thanks to my wife, Vivian, and to my employers, for their continueing support of my activities on behalf of the INTEREX conferences.

*F. Stephen Gauss
U. S. Naval Observatory
Washington, D.C.
1 August 1986*

TABLE OF CONTENTS

Tutorials

Migration Strategies.....	T 01
Gail Kinstler,Hewlett-Packard Co.	
Using Regular Expressions In EDIT/1000.....	T 02
John Johnson,Hewlett-Packard Co.	
System Programming In Fortran.....	T 03
Bill Gibbons,Markheim Systems	
Performance Data Collection and Analysis in the RTE Environment.....	T 04
Dave Glover, Nigel Clunes,Hewlett-Packard Co.	
Open System Tools.....	T 05
Kevin Morgan,Hewlett-Packard Co.	
FST- A High Performance Backup Utility For RTE.....	T 06
Bill Hassell,Hewlett-Packard Co.	
HPUX-RT.....	T 07
Bill Jacobs,Hewlett-Packard Co.	

HP1000 Papers

KERMIT- A File-transfer Utility.....	1001
Paul Schumann,E-Systems	
Remote Control From Europe Of A Telescope In South America.....	1002
Gianni Raffi,European Southern Observatory	
Software Conversion Using An Automated Development Methodology.....	1003
Charlie Small,SYSLOG	
Design Considerations for Fourth Generation Language Developement....	1004
William Gaines, Jonathan French, Industrial Computer Corp.	
A Modular Integration of Factory Cells.....	1005
Robert Combs,C & L Systems	
Creating Customized Quality Control Charts.....	1006
Cal Bonine,Statware	
Connection of Black Box Devices to the HP1000 A-series.....	1007
Wayne Asp,Hewlett-Packard Co.	
Understanding the New Serial I/O Drivers.....	1008
Johnny Klonaris,Hewlett Packard Co.	
Understanding the New Serial I/O Drivers.....	1009
Alan Tibbetts,Telos Consulting	
A Set Of IMAGE/1000 Database Tools For Screen Access.....	1010
Data Manipulation and Statistics	
Nhantu Le, Edward J. Kulis,Collagen Corp.	
CCWORD/1000 - HP1000 Word Processing.....	1011
Jens Behrens,Compuconsult A/S	
Office Automation in an HP1000 Environment.....	1012
Theresa Destra,City of Philadelphia,Air Management Services Lab	
How To Choose An Instrument Controller.....	1013
Terie Robinson,Hewlett-Packard Co.	
Software Management Strategies.....	1014
William Miller,City of Philadelphia,Air Management Services Lab	
The First HP Precision Architecture Implementation.....	1015
David Fotland,Hewlett Packard Co.	
Can Distributed Systems Be Managed Effectively?.....	1016

David Thombs, Directorate General of Defence, Ministry of Defence Establishing A Successful HP1000 Consulting Practice.....	1017
Marvin McInnis	
Databases In the Scientific And Engineering Communities.....	1018
Husni Sayed, IEM Inc.	
PC-CAD By Itself Is A Giant Step Backward.....	1019
Hector Holguin, Holguin Corp.	
The Design Of A Graphical Database For the DRAWIT Drawing System.....	1020
Marc Katz, Graphicus	
A General Purpose Process Graphics System.....	1021
Phil Walden, Hewlett-Packard Co.	
The Design of GEDIT - A General Purpose Graphics Editor.....	1022
Kurt Van Ness, Flexware Inc.	
Quality Assessment Of HP RTE Systems.....	1023
Chris Smith, Bruce Campbell, Craig Fuget, Hewlett-Packard Co.	
Designing and Implementing A Common System For the Development.....	1024
Of Large Application Packages	
Stephen Fullerton, Statware	
Effective Use Of Tools and Programming Style.....	1025
In Managing Major Software Systems	
Mathieu Federspiel, Statware	
Robotics And Data Systems In the Chemical Analysis Laboratory.....	1026
Chris Scanlon, Hewlett-Packard Co.	
Performance Analysis and Enhancements.....	1027
for a Vehicle Electrical Test System	
David Vickers, Stephen Novosad, Southwest Research Institute	
Using the Touchscreen Features of the HP150 In Application Programs..	1028
Michael Green, Kenneth Keuny, Dept of Aerospace Eng, Univ. of Maryland	
SETKY-GETKY, A Keyed Access System for the HP1000.....	1029
Dorothy Bickham, David Neumann, National Bureau of Standards	
How Do the Users Use Your System?.....	1030
Donald Wright, Interactive Computer Technology	
Using C for Portable Programming.....	1031
Tim Chase, Corporate Computer Systems	
Making RTE System Calls In HP-UX.....	1032
Grant Sidwall, Hewlett-Packard Co.	
Interfacing HP's New Tape Drives.....	1033
To HP1000 A/E/F Series	
David Doxey, Hewlett-Packard Co.	

HP9000 Papers

HP-UX: Using Standards To Solve Real World Problems.....	9001
Val Jermoluk, Chris Bego, Hewlett-Packard Co.	
Decreasing Realtime Process Dispatch Latency Through.....	9003
Kernal Preemption	
David Lennert, Hewlett Packard Co.	
Interpreters/Compilers- Their Differences and Merits.....	9004
Husni Sayed, IEM Inc.	
Disc Performance On HP-UX.....	9005
Carol Hubecka, Hewlett-Packard Co.	

Index By Author

Asp, Wayne, Hewlett-Packard Co.....	1007
Connection of Black Box Devices to the HP A-series	
Bego, Chris, Hewlett-Packard Co.....	9001
HP-UX: Using Standards To Solve Real World Problems	
Behrens, Jens, Compuconsult A/S.....	1011
CCWORD/ - HP Word Processing	
Bickham, Dorothy, National Bureau of Standards.....	1029
SETKY-GETKY, A Keyed Access System for the HP1000	
Bonine, Cal, Statware.....	1006
Creating Customized Quality Control Charts	
Campbell, Bruce, Hewlett-Packard Co.....	1023
Quality Assessment Of HP RTE Systems	
Chase, Tim, Corporate Computer Systems.....	1031
Using C for Portable Programming	
Clunes, Nigel, Hewlett-Packard Co.....	T 04
Performance Data Collection and Analysis in the RTE Environment	
Combs, Robert, C & L Systems.....	1005
A Modular Integration of Factory Cells	
Destra, Theresa, City of Philadelphia, Air Management Services Lab...	1012
Office Automation in an HP Environment	
Doxey, David, Hewlett-Packard Co.....	1033
Interfacing HP's New Tape Drives To HP A/E/F Series	
Federspiel, Mathieu, Statware.....	1025
Effective Use Of Tools and Programming Style	
In Managing Major Software Systems	
Fotland, David, Hewlett Packard Co.....	1015
The First HP Precision Architecture Implementation	
French, Jonathan, Industrial Computer Corp.....	1004
Design Considerations for Fourth Generation Language Development	
Fuget, Craig, Hewlett-Packard Co.....	1023
Quality Assessment Of HP RTE Systems	
Fullerton, Stephen, Statware.....	1024
Designing and Implementing A Common System For the Development	
Of Large Application Packages	
Gaines, William E., Industrial Computer Corp.....	1004
Design Considerations for Fourth Generation Language Development	
Gibbons, Bill, Markheim Systems.....	T 03
System Programming In Fortran	
Glover, Dave, Hewlett-Packard Co.....	T 04
Performance Data Collection and Analysis in the RTE Environment	
Green, Michael, Dept of Aerospace Eng, Univ. of Maryland.....	1028
Using the Touchscreen Features of the HP150 In Application Programs	
Holguin, Hector, Holguin Corp.....	1019
PC-CAD By Itself Is A Giant Step Backward	

Hubecka,Carol,Hewlett-Packard Co.....	9005
Disc Performance On HP-UX	
Hassell,Bill,Hewlett-Packard Co.....	T 06
FST- A High Performance Backup Utility For RTE	
Jacobs,Bill,Hewlett-Packard Co.....	T 07
HPUX-RT	
Jermoluk,Val,Hewlett-Packard Co.....	9001
The HP-UX Strategy- Using Standards To Solve Real World Problems	
Johnson,John,Hewlett-Packard Co.....	T 02
Using Regular Expressions In EDIT/	
Katz,Marc,Graphicus.....	1020
The Design Of A Graphical Database For the DRAWIT Drawing System	
Keuny,Kenneth,Dept of Aerospace Eng,Univ. of Maryland.....	1028
Using the Touchscreen Features of the HP150 In Application Programs	
Kinstler,Gail,Hewlett-Packard Co.....	T 01
Migration Strategies	
Klonaris,Johnny,Hewlett Packard Co.....	1008
Understanding the New Serial I/O Drivers	
Kulis,Edward J.,Collagen Corp.....	1010
A Set Of IMAGE/ Database Tools For Screen Access	
Data Manipulation and Statistics	
Le,Nhantu,Collagen Corp.....	1010
A Set Of IMAGE/ Database Tools For Screen Access	
Data Manipulation and Statistics	
Lennert,David,Hewlett Packard Co.....	9003
Decreasing Realtime Process Dispatch Latency Through	
Kernal Preemption	
McInnis,Marvin.....	1017
Establishing A Successful HP Consulting Practice	
Miller,William,City of Philadelphia,Air Management Services Lab...	1014
Software Management Strategies	
Morgan,Kevin,Hewlett-Packard Co.....	T 05
Open System Tools	
Neumann,David,National Bureau of Standards.....	1029
SETKY-GETKY, A Keyed Access System for the HP1000	
Novosad,Stephen,Southwest Research Institute.....	1027
Performance Analysis and Enhancements	
for a Vehicle Electrical Test System	
Raffi,Gianni,European Southern Observatory.....	1002
Remote Control From Europe Of A Telescope In South America	
Robinson,Terie,Hewlett-Packard Co.....	1013
How To Choose An Instrument Controller	
Sayed,Husni,IEM..Inc.....	1018
Databases In the Scientific And Engineering Communities	
Sayed,Husni,IEM..Inc.....	9004
Interpreters/Compilers- Their Differences and Merits	

Scanlon,Chris,Hewlett-Packard Co.....	1026
Robotics And Data Systems In the Chemical Analysis Laboratory	
Schumann,Paul,E-Systems.....	1001
KERMIT- A File-transfer Utility	
Sidwall,Grant,Hewlett-Packard Co.....	1032
Making RTE System Calls In HP-UX	
Small,Charlie,SYSLOG.....	1003
Software Conversion Using An Automated Development Methodology	
Smith,Chris,Hewlett-Packard Co.....	1023
Quality Assessment Of HP RTE Systems	
Thombs,David,Directorate General of Defence,Ministry of Defence...	1016
Can Distributed Systems Be Managed Effectively?	
Tibbetts,Alan,Telos Consulting.....	1009
Understanding the New Serial I/O Drivers	
Van Ness,Kurt,Flexware Inc.....	1022
The Design of GEDIT - A General Purpose Graphics Editor	
Vickers,David,Southwest Research Institute.....	1027
Performance Analysis and Enhancements for a Vehicle Electrical Test System	
Walden,Phil,Hewlett-Packard Co.....	1021
A General Purpose Process Graphics System	
Wright,Donald,Interactive Computer Technology.....	1030
How Do the Users Use Your System?	

Index By Title

A General Purpose Process Graphics System.....	1021
Phil Walden, Hewlett-Packard Co.	
A Modular Integration of Factory Cells.....	1005
Robert Combs, C & L Systems	
A Set Of IMAGE/1000 Database Tools For Screen Access.....	1010
Data Manipulation and Statistics	
NhanTu Le, Edward J. Kulis, Collagen Corp.	
Can Distributed Systems Be Managed Effectively?.....	1016
David Thombs, Directorate General of Defence, Ministry of Defence	
CCWORD/1000 - HP1000 Word Processing.....	1011
Jens Behrens, Compuconsult A/S	
Connection of Black Box Devices to the HP1000 A-series.....	1007
Wayne Asp, Hewlett-Packard Co.	
Creating Customized Quality Control Charts.....	1006
Cal Bonine, Statware	
Databases In the Scientific And Engineering Communities.....	1018
Husni Sayed, IEM Inc.	
Decreasing Realtime Process Dispatch Latency Through.....	9003
Kernal Preemption	
David Lennert, Hewlett Packard Co.	
Design Considerations for Fourth Generation Language Developement....	1004
William Gaines, Jonathan French, Industrial Computer Corp.	
Designing and Implementing A Common System For the Development.....	1024
Of Large Application Packages	
Stephen Fullerton, Statware	
Disc Performance On HP-UX.....	9005
Carol Hubecka, Hewlett-Packard Co.	
Effective Use Of Tools and Programming Style.....	1025
In Managing Major Software Systems	
Mathieu Federspiel, Statware	
Establishing A Successful HP1000 Consulting Practice.....	1017
Marvin McInnis	
FST- A High Performance Backup Utility For RTE.....	T 06
Bill Hassell, Hewlett-Packard Co.	
How Do the Users Use Your System?.....	1030
Donald Wright, Interactive Computer Technology	
How To Choose An Instrument Controller.....	1013
Terie Robinson, Hewlett-Packard Co.	
HPUX-RT.....	T 07
Bill Jacobs, Hewlett-Packard Co.	
Interfacing HP's New Tape Drives.....	1033
To HP1000 A/E/F Series	
David Doxey, Hewlett-Packard Co.	
Interpreters/Compilers- Their Differences and Merits.....	9004
Husni Sayed, IEM Inc.	
KERMIT- A File-transfer Utility.....	1001
Paul Schumann, E-Systems	
Making RTE System Calls In HP-UX.....	1032
Grant Sidwall, Hewlett-Packard Co.	
Migration Strategies.....	T 01
Gail Kinstler, Hewlett-Packard Co.	

Office Automation in an HP1000 Environment.....	1012
Theresa Destra, City of Philadelphia, Air Management Services Lab	
Open System Tools.....	T 05
Kevin Morgan, Hewlett-Packard Co.	
PC-CAD By Itself Is A Giant Step Backward.....	1019
Hector Holguin, Holguin Corp.	
Performance Analysis and Enhancements.....	1027
for a Vehicle Electrical Test System	
David Vickers, Stephen Novosad, Southwest Research Institute	
Performance Data Collection and Analysis in the RTE Environment.....	T 04
Dave Glover, Nigel Clunes, Hewlett-Packard Co.	
Quality Assessment Of HP RTE Systems.....	1023
Chris Smith, Bruce Campbell, Craig Fuget, Hewlett-Packard Co.	
Remote Control From Europe Of A Telescope In South America.....	1002
Gianni Raffi, European Southern Observatory	
Robotics And Data Systems In The Chemical Analysis Laboratory.....	1026
Chris Scanlon, Hewlett-Packard Co.	
SETKY-GETKY, A Keyed Access System for the HP1000.....	1029
Dorothy Bickham, David Neumann, National Bureau of Standards	
Software Conversion Using An Automated Development Methodology.....	1003
Charlie Small, SYSLOG	
Software Management Strategies.....	1014
William Miller, City of Philadelphia, Air Management Services Lab	
System Programming In Fortran.....	T 03
Bill Gibbons, Markheim Systems	
The Design Of A Graphical Database For the DRAWIT Drawing System....	1020
Marc Katz, Graphicus	
The Design of GEDIT - A General Purpose Graphics Editor.....	1022
Kurt Van Ness, Flexware Inc.	
The First HP Precision Architecture Implementation.....	1015
David Fotland, Hewlett Packard Co.	
HP-UX: Using Standards To Solve Real World Problems.....	9001
Val Jermoluk, Chris Bego, Hewlett-Packard Co.	
Understanding the New Serial I/O Drivers.....	1008
Johnny Klonaris, Hewlett Packard Co.	
Understanding the New Serial I/O Drivers.....	1009
Alan Tibbetts, Telos Consulting	
Using C for Portable Programming.....	1031
Tim Chase, Corporate Computer Systems	
Using Regular Expressions In EDIT/1000.....	T 02
John Johnson, Hewlett-Packard Co.	
Using the Touchscreen Features of the HP150 In Application Programs..	1028
Michael Green, Kenneth Keuny, Dept of Aerospace Eng, Univ. of Maryland	

Migration Strategies

by: Kinstler, Gail

We regret that this paper
was not received for
inclusion in these proceedings.

Using Regular Expressions in EDIT/1000

by: Johnston, John

We regret that this paper
was not received for
inclusion in these proceedings.

System Programming in Fortran

Bill Gibbons
Mirkheim Systems
P.O. Box 203
Los Altos, CA 94023-0203
U.S.A.

1. Introduction

The main difference between system programming and applications programming is *control*. The system programmer wants to control implementation details such as:

- Data representation
- Memory organization
- Low-level algorithms
- File I/O
- Device I/O
- System resources

Most computer languages, including Fortran, give you very little control in these areas. Some don't even provide I/O; it is tacked on as an afterthought.

Beginning programmers are told not to worry about implementation details; without doubt, this makes programming much easier - if you never write system programs. In the real world, we often must worry about these details if we want to write useful programs.

The Fortran compiler and runtime libraries for the HP1000 provide a number of extensions for system programming. Many of these are nonportable; some are outright dangerous. However, they always result in more readable programs than assembly language.

In particular, this paper covers:

- Bit pushing
- Byte arrays
- Nonstandard calling sequences
- Nonstandard common blocks
- Freespace use
- Custom EMA mapping
- Strings and tables in code space
- Setjmp and Longjmp calls
- Various I/O extensions

2. Bit & Byte Manipulation

2.1 Bit Pushing

The traditional way to manipulate bits in Fortran is with integer arithmetic. Bit fields can be extracted with divide and MOD; they can be combined with multiply and add. For example, the following code extracts the middle eight bits of a word:

```
I = MOD(J/16,256)
```

This works fine for positive numbers, but fails for negative numbers. The problem is that division by a power of two is not the same as a right shift for negative numbers. Consider $(-3)/(2) = (-1)$, where (-3) is 1101 and (-1) is 1111, but (-3) right shifted one is 1110 or 0110, depending on whether you extend the sign bit. This problem occurs on all two's complement machines such as the HP1000.

The MOD function also produces an unexpected result for negative numbers. The remainder of dividing a negative number by a positive number is negative or zero. For instance, $\text{MOD}(-3, 2) = (-1)$.

On the other hand, multiplying by a power of two works fine. There is a potential problem when shifting a field all the way to the left, e.g. multiplying by 256 to form the first character in a word. If the sign of the result should become set, the multiply actually overflows. On the HP1000, the overflow is ignored and the least 16 bits of the result are used; this makes the above case work.

Using an "add" to combine fields works if they are disjoint; but if two one-bits are added in the same position, they cause a carry-out which invalidates the result. This makes "add" a poor substitute for an "or".

Subtract can be used to zero out a field, but it is cumbersome. To zero out the middle 8 bits of a word, you have to write:

```
I = J - MOD(J/16,256)*16
```

What we really want is the bitwise logical operations: AND, OR, exclusive OR, etc. In a standard Fortran program, you can use the .AND., .OR., .EQV., .NEQV. and .NOT. operators on LOGICAL variables. Many compilers (including FTN7X) implement these by operating on the whole word. With EQUIVALENCE, you can get bitwise logical operations on integer data.

This has two serious drawbacks; it is hard to write and nonportable. Some compilers (again including FTN7X) let you use these operators on integer data. This eliminates the EQUIVALENCE, and produces very readable code; it is still somewhat nonportable. It is more portable than the EQUIVALENCE solution, which may compile but give wrong answers on some machines; if the second solution compiles, it will almost certainly work.

Unfortunately, the precedence of the logical operators is confusing when you use them on integer data. For example,

```
IF (IJK .AND. 3 .EQ. 0) ...
```

is parsed as

```
IF (IJK .AND. (3 .EQ. 0)) ...
```

because .EQ. has a higher precedence. This statement causes an error: mixing integer and logical data. It is always a good idea to use parentheses when writing such *masking expressions*.

Actually, though you may see the above techniques in old programs, you should not use them anymore. There is a set of functions known as MIL-STD-1753 which provide almost all the bit manipulation you need. Many compilers implement this standard, which makes bit pushing almost portable. FTN7X provides them for both 16-bit and 32-bit integers. Where practical, the functions are done with inline code instead of library routine calls.

The MIL-STD-1753 masking functions perform bitwise logical operations. They are IAND, IOR, IXOR, and NOT. For 16-bit integers, FTN7X uses inline code to implement these functions. For example:

```
I = J .AND. 377B    yields:    LDA J
                                AND =B377
                                STA I
```

The ISHFT and ISHFTC functions perform logical and circular shifts, respectively. A positive shift count indicates a left shift; negative, a right shift. The ISHFTC call includes the size of the field to shift; for example, ISHFTC(I,4,8) swaps the nibbles in the low byte of I. When ISHFT is used with a constant shift count, FTN7X generates inline code. To get inline code for ISHFTC you must also have a field size of 16 or 32. (Other field sizes work, but a library routine is used.) For example:

```
I = ISHFT(J,-4) .AND. 377B    yields:    LDB J
                                           LSR 4
                                           LDA B
                                           AND =B377
                                           STA I
```

which is much better than the previous example using divide and MOD. Since fetching a bit field is such a common operation, MIL-STD-1753 also includes the IBITS function, which does just that. The field is described by the rightmost bit number and the number of bits. For example, we could rewrite the above code as:

```
I = IBITS(J,4,8)            which yields:    LDA J
                                           RAR,RAR
                                           RAR,RAR
                                           AND =B377
                                           STA I
```

The generated code is better, too, since it does not use the B-register; in the earlier example, any value in the B-register would be stored into a temp.

The special case of a one-bit field is so common that there are special functions to set, clear, and test a bit, namely IBSET, IBCLR and BTEST. The second parameter is the bit number. For example,

```
IF (BTEST(J,0)) ...    which yields:    LDA J
                                           SLA,RSS
                                           JMP <around>
```

tests whether J is odd. Note that constant bit numbers generate inline code, and that BTEST returns a LOGICAL result.

The last MIL-STD-1753 routine is the MVBITS subroutine. This routine copies a bit field in one variable into a field of the same size in another variable. The call specifies both fields; think of writing the source and destination IBITS parameters together, but leaving out the second field size (which is redundant). For example:

```
CALL MVBITS(I, 0, 4, J, 12)
```

moves the low 4 bits of I into the high 4 bits of J.

The designers of MVBITS made a small design error; it should have been a function instead of a subroutine. If it were actually done as a subroutine, there would be no way to handle mixed single and double integers. The FTN7X compiler changes the call into a function, as in:

```
J = .IMBS(I, 0, 4, J, 12)
```

which is always well-defined. If I and J have mixed types, the shorter one is converted to double integer and the double integer routine is called. The result is a double integer, which is shortened if necessary.

2.2 Byte Arrays

One data type that system programmers want in Fortran is bytes. Some Fortran compilers provide a byte data type as `INTEGER*1` or `LOGICAL*1`; the only byte type in `FTN7X` is `CHARACTER*1`.

Character variables may seem an odd way to store bytes of numeric data, but it can be quite useful. In practice, character data is implemented as a byte data type anyway. You can't store numbers directly into a character variable or perform arithmetic, but you can convert between characters and integers.

The two conversion routines `CHAR` and `ICHAR` convert between integer and character data. No actual conversion is done; only the meaning of the value changes. For example, `ICHAR('A')` has the value 101B, which is the bit pattern used to represent 'A'. The data itself is unchanged; only the meaning is different. This holds for all 256 characters, so all integers in the range 0-255 can be stored in character variables.

For example, a large table of small constants can be kept in a character array. One example is the state transition table generated by the lexical scanner generator `LEX`. The state table would be coded in assembly language:

```
      ENT TABLE
TABLE EQU *
      BYT 12,34,56,0,7,7,7,4,17
      BYT 17,17,  -etc-
```

The Fortran program would access the table as a common block:

```
$ALIAS /TABLE/, NOALLOCATE
      ...
COMMON /TABLE/ TABLE(1000)
CHARACTER*1 TABLE
      ...
STATE = ICHAR(TABLE(J))
```

Which references the J^{th} entry of the table. It takes a few more instructions to access a character array than to access an integer array. In this example, the table is large and the access is only done once or twice, so the net savings is large.

Another use of character arrays is to scan binary data. If the data is organized as bytes but does not honor word boundaries, character arrays can be used to scan the record.

For example, if an input record from a device has the format:

```
byte 1: device ID
byte 2: device status
byte 3: channel number
byte 4: first byte of value
byte 5: second byte of value
```

The data could be extracted from the record like this:

```
CHARACTER RECORD(5)
READ(LU) RECORD
ID = ICHAR(RECORD(1))
STATUS = ICHAR(RECORD(2))
CHANNEL = ICHAR(RECORD(3))
VALUE = ISHFT(ICHAR(RECORD(4)),8) + ICHAR(RECORD(5))
```

This could be made clearer using EQUIVALENCE statements:

```

CHARACTER RECORD(5), CID, CSTATUS, CCHANNEL
EQUIVALENCE (RECORD(1),CID)
EQUIVALENCE (RECORD(2),CSTATUS)
EQUIVALENCE (RECORD(3),CCHANNEL)
EQUIVALENCE (RECORD(4),VALUE)
READ(LU) RECORD
ID = ICHAR(CID)
STATUS = ICHAR(CSTATUS)
CHANNEL = ICHAR(CCHANNEL)

```

Note that we equivalenced RECORD(4) and VALUE directly, since the data was already two bytes long. To honor this equivalence, the compiler had to start RECORD at an odd byte boundary. If RECORD was in common or had other equivalences, this might not be possible. In that case we could write:

```

CHARACTER OVALUE*2, EVALUE*2
EQUIVALENCE (RECORD(4),OVALUE)
EQUIVALENCE (EVALUE,VALUE)
...
EVALUE = OVALUE

```

which moves the two bytes from OVALUE, which could start on an odd byte, into EVALUE, which is not restricted and can be equivalenced to VALUE.

When you pass a character variable or array to a subroutine, you can change its *shape* by declaring a different length, number of dimensions, and dimension sizes (this is allowed by the standard). For example, the following two subroutine reverse a string. The first one uses ordinary substrings:

```

SUBROUTINE REVERSE(S)
CHARACTER S(*), T
L = LEN(S)
DO I=1,L/2
  T = S(I:I)
  S(I:I) = S(L-I+1:L-I+1)
  S(L-I+1:L-I+1) = T
ENDDO
END

```

The second one treats the string as an array. Since the length of the string is ignored, we must pass it explicitly:

```

SUBROUTINE REVERSE(S, L)
CHARACTER S(*)*1, T
DO I=1,L/2
  T = S(I)
  S(I) = S(L-I+1)
  S(L-I+1) = T
ENDDO
END

```

The first routine compiles to 146 words of code and makes four calls to a library routine in each loop. The second routine compiles to 78 words and does everything inline using the LBT and SBT instructions.

3. Aliasing (Nonstandard External Symbols)

The ALIAS capability in F77N lets you use subroutines and common blocks with nonstandard names and other attributes. The alternate name is supplied as a Fortran string, so it can have special characters, lower case, and even blanks. The attributes are specified with keywords, as shown in the examples below.

3.1 Subprogram Aliasing

There are a number of interesting system library routines that have nonstandard names and/or calling sequences. Of course, system programmers want to use these routines. Some of the machine instructions also look like subroutines, and the ALIAS mechanism can be used to call some of them directly.

Many assembly language routines use the *direct* calling sequence, which does not have the "DEF return" word. This is different from the standard .ENTR calling sequence which Fortran uses, but you can tell Fortran to use direct calls with ALIAS:

```
$ALIAS MAPIT = '.LBPR', DIRECT
```

In this example, the routine is actually a machine instruction which performs EMA/VMA mapping.

Another nonstandard convention is the EXEC-style *alternate return*. Certain system routines, including EXEC, may return to the normal return address when there is an error, and to the next word if there is no error. This feature maps conveniently into the Fortran alternate return feature, except that the actual Fortran implementation is completely different. With ALIAS, you can make the compiler generate its alternate returns using the EXEC convention:

```
$ALIAS LURQ, NOABORT  
CALL LURQ(40001B, 6, 1, *99)
```

will return to statement 99 if there is an error. The compiler knows about EXEC, XLUEX, DEXEC and REIO; the ALIAS is not necessary for these routines.

3.2 Common Block Aliasing

A common block ALIAS is specified with slashes around the name, as in:

```
$ALIAS /TIME/ = '$TIME', NOALLOCATE
```

The NOALLOCATE option is used to make the common block name an *external symbol* (or *entry*, in a block data) instead of an *allocate* symbol. Allocate symbols may not work with system entry points such as \$TIME because they are never undefined: as soon as the linker sees an allocate symbol, it defines it.

Using the two kinds of ALIAS, you can rummage around in the operating system at will. For example:

```
$ALIAS /TIME/ = '$TIME', NOALLOCATE  
$ALIAS LOAD = '.XLA', DIRECT  
COMMON /TIME/ TIME(2)  
INTEGER TIME, HIGH, LOW  
LOW = LOAD(TIME(1))  
HIGH = LOAD(TIME(2))
```

This code fetches the current system clock. There are many such symbols in the operating systems which might interest the system programmer.

There are times when you want to access a memory location by its address. With a common block ALIAS, you can access specific addresses. For example:

```
$ALIAS /ID/ = 1717B  
COMMON /ID/ ID
```

This code lets you access your program's ID segment address (on RTE-6). More generally,

```
$ALIAS /MEM/ = 0  
COMMON /MEM/ MEM(0:0)
```

makes the array element MEM(J) the same as location J.

4. Freespace

The freespace area, or heap, is the area between the end of your program and the end of the memory partition. You can adjust the size of freespace with the CI command **SZ** or the linker command **HE**, without reloading the program. This is very convenient for certain kinds of programs.

You need two things to use freespace: the size of freespace available, and an access method. The size can be found with **LIMEM**; there are several access methods.

The simplest access is with blank common. In a non-CDS program, **LINK** puts blank common just before freespace. If the last (or only) item in blank common is a one-element array, you can use the array to get to freespace. This only works in non-CDS programs; in CDS programs the stack is between blank common and freespace.

In a similar manner, you can arrange your **LINK** command file to put a named common block at the end of your program. You must use the **NOALLOCATE** option for the common block; this keeps **LINK** from defining it immediately. After the libraries have been searched you relocate the **BLOCKDATA** for the common block so that it becomes the last module in the load. In this case, you must *not* use blank common because **LINK** will put it after the last module. Again, this method does not work with CDS because of the stack.

Other methods involve using the address of freespace, which you can get with **LIMEM**. If you know the address of an array **M**, you can access any given address **I** with the expression **M(I-addr(M)+1)**. As a quick proof, note that if **I** is the address of **M** then **M(I-addr(M)+1) = M(1)**. By declaring **M** with a lower bound of zero, you can eliminate the offset of one. For example:

```
INTEGER ADDRESSOF, M(0:0), I, N
N = ADDRESSOF(M)
WRITE(1,*) 'Contents of location', I, ' = ', M(I-N)
```

On older systems without **AddressOf**, the routine **.DRCT** serves the same purpose. On such a system, the above program must have the line:

```
$ALIAS ADDRESSOF = '.DRCT', DIRECT
```

The above method has the advantage that it works on a variety of different machines. On the other hand, the use of the alias:

```
$ALIAS /MEM/ = 0
```

is less portable but more readable, since there is no constant offset to subtract. The **ALIAS** method also produces slightly more efficient code. In either case, memory should be allocated by a central routine which returns a (possibly adjusted) subscript. This makes it easy to port the code to another machine. On a machine with no freespace mechanism or no way to use it from Fortran, you could just declare **MEM** as a large array and allocate space from the array itself.

5. Custom EMA Mapping

When you need more than a few thousand words of memory for data, you have to use **EMA**. The good news is that you can run a program that uses megabytes of data. The bad news is that it may be very slow.

The problem with **EMA** is that your program changes the map registers for *each and every EMA access*. There is no way for the compiler to map a large amount of data and leave it mapped in while you work on it. However, you can do exactly that with custom mapping. (This section of the paper assumes that you have some knowledge of how **EMA** works already.)

There are two parts to custom mapping. First, you must change the map registers. Second, you have to get to the data on the mapped pages of memory.

There are a number of EMA mapping instructions; some are callable from Fortran, some are not. The most useful are .LBPR and .ESEG .

The Fortran compiler uses .LBPR to map simple variables. It takes one parameter, an EMA address. The page containing that location, and the next page, are mapped in. The instruction sets the B-register to the mapped address which corresponds to the EMA address. For example, given:

```
$ALIAS /MEM/ = 0
$ALIAS MAPPER=' .LBPR', DIRECT
COMMON /MEM/ MEM(0:0)
INTEGER*4 MAPPER
EMA LARGE(100000)
```

the sequence:

```
J = MAPPER(LARGE(I))
K = MEM(J)
```

does the same thing as the sequence:

```
K = LARGE(I)
```

Note that the first statement does an implicit conversion from double integer (MAPPER result) to single integer (J). Since double integer functions return the result in (A,B), and the conversion just uses the low-order word, we are putting (B) into the variable J. This is the mapped address.

This example isn't very practical because the compiler can handle simple mapping by itself. But suppose we wanted to zero out an array in EMA. We might code the routine:

```
$ALIAS /MEM/ = 0
$ALIAS MAPPER = ' .LBPR', DIRECT
SUBROUTINE CLEAR_EMA(ARRAY, SIZE)
IMPLICIT NONE
EMA ARRAY
INTEGER ARRAY(*), SIZE
COMMON /MEM/ MEM(0:0)
INTEGER MEM
INTEGER INDEX, LEFT, ADDR, CHUNK
INTEGER*4 MAPPER
LEFT = SIZE
INDEX = 1
DO WHILE (LEFT .GT. 0)
  ADDR = MAPPER(ARRAY(INDEX))
  CHUNK = MIN(LEFT, 1025)
  DO ADDR=ADDR, ADDR+CHUNK-1
    MEM(ADDR) = 0
  ENDDO
  INDEX = INDEX + CHUNK
  LEFT = LEFT - CHUNK
ENDDO
END
```

Note that at least 1025 words are mapped on each call. With a little care, we could use 2048 words from each call except the first and last.

Sometimes it's best to use actual pages and MSEG addresses. A normal 2-page MSEG begins at address 74000B; we can use a common block ALIAS to access it as an array. The pages of EMA can be arranged as columns of an array. For example:

```
$ALIAS /MSEG/ = 74000B
$ALIAS MAPPER = '.LBPR', DIRECT
$EMA /EMA/
    SUBROUTINE MAPIT
    COMMON /EMA/ EMA(0:1023,0:99)
    COMMON /MSEG/ MSEG(0:2047)
    INTEGER EMA, PAGE, OFFSET
    ...
    CALL MAPPER(EMA(0,PAGE))
    I = MSEG(OFFSET)
```

This routine uses (page,offset) pairs to access locations in EMA. Note that there must only be one array in EMA, since we are assuming that it starts at EMA location zero.

The .ESEG routine provides another way to map using page numbers. It is passed an array of page numbers; it maps corresponding pages of the MSEG to point to these pages. .ESEG must be called from assembly language; see the Programmer's Reference Manual.

The .ESEG instruction is much more powerful than .LBPR. It can map more than two pages at a time, and they need not be contiguous. This opens up a number of possibilities; in particular, a program can work on data in different parts of EMA simultaneously without remapping each time. This is the main purpose of .ESEG; the EMA Vector Instruction Set uses it to handle multiple vectors in EMA.

One disadvantage of EMA is that the two-page MSEG comes out of your 32-page data segment, so your non-EMA data is two pages smaller. In some cases these two pages can be reclaimed.

If you ask for two more pages of EMA than you need, you can map these pages into the MSEG and leave them mapped in. When you use other areas of EMA, you must map the extra pages back when you're done. The program uses the extra pages as additional freespace; it is totally transparent that they happen to be in EMA.

A good use of this method is for putting DCB's in EMA. The access to the DCB's is tightly controlled (only in FMP calls), so you have a small number of places where mapping and re-mapping must be done. Of course, the data in the extra pages can't be used during any mapped FMP calls, because the MSEG is temporarily different.

6. Strings and Tables in Code Space

In CDS programs, space in the data segment is at a premium. Large arrays can be moved into EMA, but you can't initialize EMA variables in a DATA statement (for the moment). This makes it difficult to move large constant tables out of the data segment. However, with a little work, large tables (including strings) can be moved into a code segment.

The trick to putting constants in code space is the cross-map load instruction .XLA2 and the cross-map move instructions .MW20 and .MB20. The load instruction works like a cross-load, but it fetches the word from the current code segment. The move instructions work like the CDS move-words instruction .MW00, but they move data from the current code segment into the data segment.

These instructions must be used in assembly language. Since there is no way to get Fortran to put data in code space, the data itself must also be in assembly language. Also, the cross-map instruction and the data must be in the same code segment; the best way to ensure this is to put both the code and the data in the same module (.NAM).

Actually, the cross-load can be done from Fortran too. For example, the data in:

```
MACRO
    NAM TABLE
    CDS ON
    ENT TABLE
    RELOC CODE
TABLE    DEC 1,2,3,4,5
END
```

could be referenced by the Fortran subroutine:

```
$ALIAS /TABLE/, NOALLOCATE
$ALIAS LOAD = '.XLA2', DIRECT
INTEGER FUNCTION CGET(I)
COMMON /TABLE/ ITABLE(5)
CGET = LOAD(ITABLE(I))
END
```

The compiler thinks that `TABLE` is in data space. It performs arithmetic to get the address of `ITABLE(I)` without ever trying to fetch the value. It passes this address to `LOAD`, which is really `.XLA2`; the data is loaded to `(A)`, which is where function results are returned.

Either way, the code that accesses the table should be centralized, as in the `CGET` routine above.

7. Setjmp and Longjmp

When an error occurs many subroutine calls deep in a program, it's useful to just pop out of the current subroutine into some high-level routine, without executing all the `RETURN` statements in between. In non-CDS programs, you can do this just by calling the high-level routine. The high-level routine can no longer do a `RETURN`, but otherwise the program can continue normally.

In a CDS program, such calls are recursive. This has two drawbacks: all the local variables will be new, and some stack space has been lost. That makes this approach almost useless.

UNIX systems have a solution to this problem; they supply two routines called `SETJMP` and `LONGJMP`. The `SETJMP` routine records the current program counter *and* the stack pointer; the `LONGJMP` routine restores them.

This means that there are two ways to get to the statement after a `SETJMP` call: returning from the call itself, or resuming after a `LONGJMP` call. The two are distinguished by making `SETJMP` a function which returns zero; when `LONGJMP` is called, it looks as if `SETJMP` returned with a nonzero value.

You can do the exact same thing with CDS programs using the `SETJMP` and `LONGJMP` on the next page.

To use `SETJMP`, call it with a 3-word buffer. When you need to get back, call `LONGJMP` with the same buffer and a nonzero value. For example:

```
INTEGER ENV(3), SETJMP
COMMON ENV
...
IF (SETJMP(ENV).EQ.0) THEN
    <first time>
ELSE
    <from LONGJMP>
ENDIF
```

With a LONGJMP call such as:

```

    INTEGER ENV(3)
    COMMON ENV
    ...
    CALL LONGJMP(ENV,1)

```

The alternative to SETJMP and LONGJMP is to have the routine return an error code. Then each routine that calls it must check for the error, and return an error code, and so on. The SETJMP solution is unstructured, but it makes some programs much cleaner.

The SETJMP and LONGJMP code:

```

    NAM SETJMP
    CDS ON
    ENT SETJMP,LONGJMP
    EXT .CCQA,.XLA2,.EXITO

*
    RELOC LOCAL
RTN_ADDR EQU *-4      OUR RETURN ADDRESS AT Q+2
CST_PCNT EQU *-3      OUR RETURN SEGMENT AT Q+3
ENVIRON  BSS 1        3-WORD ENVIRONMENT BUFFER
RESULT   BSS 1        RESULT PASSED TO LONGJMP

*
    RELOC CODE
SETJMP   ABS 7
        LDA RTN_ADDR    DID WE GET HERE WITH A
        CCE,SSA         CROSS-SEGMENT CALL ?
        JMP SETJMP2     YES. SEGMENT NUMBER VALID.

*
        RAL,ERA         NO. FAKE IT AS IF WE DID:
        STA RTN_ADDR    SET THE CROSS-SEGMENT BIT
        JSB .XLA2       GET OUR SEGMENT NUMBER.
        DEF 2000B
        STA CST_PCNT    AND SET AS THE RETURN SEGMENT.

*
SETJMP2  JSB .CCQA       (A) = OUR STACK POINTER
        DLD @A          (B) = PREVIOUS STACK POINTER
        STB @ENVIRON    SAVE IT
        ISZ ENVIRON
        DLD RTN_ADDR    (A,B) = RETURN ADDRESS & SEG
        DST @ENVIRON    SAVE THEM
        CLA            RETURN ZERO.
        JSB .EXITO

*
LONGJMP  ABS 8
        JSB .CCQA       (A) = OUR STACK POINTER
        LDB @ENVIRON    (B) = SAVED STACK POINTER
        DST @A          SET IT AS OUR PREVIOUS ONE
        ISZ ENVIRON
        DLD @ENVIRON    (A,B) = RETURN ADDRESS & SEG
        DST RTN_ADDR    SET UP FOR OUR RETURN
        LDA @RESULT     RETURN 2ND PARAM AS RESULT
        JSB .EXITO     RETURN AS IF FROM SETJMP CALL

*
    END

```

Note the cross-load from location 2000 in the current code segment; this location contains the current segment number in the high byte. Since the LONGJMP call can be in a different segment, we must set up for a cross-segment return, even if the call to SETJMP was an intra-segment call.

8. Device I/O

8.1 READ and WRITE with 8-bit LU's

Most applications can get by with I/O to the terminal (LU 1), disc files, and commonly used devices such as a tape drive. These devices generally have LU numbers in the range 1-63, and Fortran READ and WRITE statements work fine with them. System programs, though, may need to use all LU's. For example, non-session programs in RTE-6 don't have an SST, so even the terminal LU may be greater than 63.

Older versions of RTE used a 6-bit LU number; high-order bits in the same word were used as option or *control bits*. When a Fortran program used a READ or WRITE, Fortran passed the entire Fortran LU to RTE, which used the lower 6 bits as a system LU and the upper bits as control bits. For example, if the line printer was LU 6, writing to LU 134 (206B) caused the first character to print instead of begin used for carriage control.

Then RTE was changed to allow LU's up to 255. Because many existing Fortran programs used control bits, the 8-bit LU was made an option in Fortran. The default is 6-bit LU's, but you can easily change or override the default to get 8-bit LU's.

The key is the symbol Z\$CWD in the file %FRPLS; see the Fortran compiler files "FTN7X and &FRPLS for details.

8.2 Binary WRITE to the Terminal

When writing an escape sequence to a terminal, you usually don't want the trailing carriage return and linefeed. Common practice on HP1000's is to end the line with an underscore; this tells the driver to omit the carriage return and linefeed.

A better and more portable method is to use binary WRITE. For example:

```
WRITE(1) CHAR(27), 'H', CHAR(27), 'J'
```

writes exactly four characters to the terminal. Note that the Escape character is easily written with the CHAR function. Better yet, put the Escape in a PARAMETER statement:

```
CHARACTER ESC  
PARAMETER (ESC=CHAR(33B))  
...  
WRITE(1) ESC, 'A'
```

Making the entire escape sequence a named constant is more clear and generates slightly more efficient code:

```
CHARACTER*(*) BLINKING  
PARAMETER (BLINKING = CHAR(27) // '&dA')  
...  
WRITE(1) BLINKING
```

Only the characters in the WRITE are written; there are no trailing blanks, CR/LF, blank-padding of odd-length records, or other extraneous characters. The binary control bit is set in the REIO call.

8.3 Reading Variable-Length Records

Applications usually deal in ASCII files, but system programs often read and write binary files. Fortran has a binary `READ` and `WRITE`, but the `READ` must know the length of the record *before* it reads it. With many files, such as relocatables, there is no way to know the record length in advance.

The solution is to try to read a large record, trap the "record-too-small" error, and find out how much was actually read. The error code is 496. The function `ITLOG` returns the number of bytes actually read in the last `READ` statement. For example:

```
INTEGER BUFFER(129)
...
READ(8, IOSTAT=IOS) BUFFER
IF (IOS .EQ. 0) STOP 'Record too big.'
IF (IOS .LT. 0) STOP 'At EOF'
IF (IOS .NE. 496) STOP 'I/O error.'
LENGTH = ITLOG() / 2
```

The above code is suitable for reading relocatable files, which have records no larger than 128 words long. Note the extra word in the buffer, which is used to catch any records which are too large.

8.4 READ/WRITE with a Z-buffer

The RTE `EXEC` read and write calls have an option called a Z-buffer. This is a second I/O buffer which is passed to the driver; the use of the Z-buffer depends on the driver. For example, the terminal drivers write the Z-buffer before they read or write the ordinary record.

This is very useful for writing to a device which will respond immediately with data which needs to be read. If the read is not posted soon enough, the data is lost. If you code the write and the read separately, RTE may interrupt your program between the two, causing the read to be posted too late to catch the incoming data.

A simple example is the terminal status request. If you send an HP terminal the escape character and a circumflex (^), it responds with status information. The read must be posted immediately after the write, or the incoming data will cause a prompt.

The ideal solution is to combine the `WRITE` and `READ` into a single request using a Z-buffer. In `FTN7X`, the `ZBUF` and `ZLEN` keywords can be used to specify a Z-buffer. For example:

```
INTEGER PROMPT
CHARACTER STRING*20
PROMPT = ISHFT(33B,8) + ICHAR('^')
READ(1, '(a)', ZBUF=PROMPT, ZLEN=-2) STRING
WRITE(1,*) STRING
```

If you try this program, you'll see the status information twice: once as it is echoed by the driver (as if it were typed in), and once from the `WRITE` statement. This is because the `READ` statement sets the *echo bit* in the read request.

There are two ways to turn off the echo bit in Fortran. One way is to use a binary `READ`. Unfortunately, the Fortran library will attempt to read its full binary buffer size (120 characters). The terminal driver will wait for 120 characters to be sent before it completes the read request. If you know exactly how many characters are coming back (and it's an even number), you can use `LGBUF` to set the buffer size; however, this is somewhat risky.

The new serial drivers in RTE provide a much better solution. You can explicitly ask for *CPU-to-CPU protocol*. When the driver uses this protocol, a Fortran program using `ZBUF` and `ZLEN` can do I/O to an intelligent device using serial I/O without the problems mentioned above.

At least one paper describing these drivers was to be presented at the conference.

8.5 HPIB and PRAM3/PRAM4

Many I/O drivers will accept one or two words of additional information in the EXEC request; these are called PRAM3 and PRAM4, because they are the third and fourth data parameters in the EXEC request. In a Fortran program, these values can be specified in a READ or WRITE:

READ(100:IPRAM3:IPRAM4) HPIP_DATA

The PRAM4 value is optional; in the above example, an LU of 100:IPRAM3 would be the same as 100:IPRAM3:0.

For HPIB, these values are the secondary and tertiary HPIB addresses. Other drivers use them in other ways. If you write your own drivers, you can pass these values directly from your Fortran program.

9. Miscellaneous I/O Extensions

There are a number of other extensions and library routines for device and file I/O. Rather than describe them all in detail, I've provided a brief summary of each. For details, see the Fortran manual and my recent TC Interface columns on Program Development.

FPOST - post buffered data to the disc immediately.

FLOCF - get the byte position and record number of a file.

FPOSN - set the byte position and record number of a file.

ITYPE - get the file type of an open file.

USE keyword - specify if the file is to be opened as shared exclusive access, or update mode.

BUFSIZ keyword - specify the packing buffer (DCB) size.

FREESPACE keyword - allocate packing buffers (DCB's) out of freespace (on the FILES directive).

NFIOB - get the number of available packing buffer blocks.

LGBUF - supply a new (possibly large) record buffer for READ and WRITE.

FFRCL - specify the maximum record size for free-field WRITE.

ITLOG - get the number of characters actually read by the last READ.

ISTAT - get the status word from the last READ or WRITE; usually meaningless after a WRITE.

10. Conclusion

The above techniques are not intended for ordinary programs. But when your choice is using these extensions or writing in assembly language, the choice is clear: the program will be more readable and maintainable written with unusual Fortran code than with straight assembly language.

Performance Data Collection and Analysis in the RTE Environment

**by: Glover, Dave
Clunes, Nigel**

We regret that this paper
was not received for
inclusion in these proceedings.

Open System Tools

by: Morgan, Kevin

We regret that this paper
was not received for
inclusion in these proceedings.

FST — A High Performance Backup Utility for RTE

**by: Hassell, Bill
Clunes, Nigel**

We regret that this paper
was not received for
inclusion in these proceedings.

HPUX-RT

by: Jacobs, Bill

We regret that this paper
was not received for
inclusion in these proceedings.

KERMIT - a file-transfer utility
Paul Schumann
E-Systems, Inc.
P.O. Box 1056 CBN 101
Greenville, TX 75401

Introduction

One of the problems we face increasingly these days is the need to transfer text and other information between similar systems with no compatible removable media, or unlike systems, whether PC's or mainframes. In many cases networking solutions to these problems either don't exist, or they are incompatible, like Ethernet vs. IEEE 802.3. If a compatible networking solution can be found, it is probably too expensive in hardware costs alone for the kind of occasional access required.

KERMIT was developed for the purpose of filling this gap using one of the cheapest and most universal "media" available - the telecommunications (IEEE RS-232) port. It uses an asynchronous, half-duplex protocol and the ASCII character-set. The protocol can be implemented in many languages and it is already available for a large variety of mainframe- to micro-size machines.

This paper is intended to serve three purposes:

- 1) Examine the KERMIT protocol, showing some of it's capabilities and limitations,
- 2) Compare KERMIT to other file-transfer methods,
- 3) Provide a manual to assist the users of KERMIT-RTE.

Section 1: The KERMIT protocol - a brief description

This is not intended as a rigorous description of the KERMIT protocol; rather, it is intended to answer some of the "why" questions about it. A full definition may be found in the KERMIT Protocol Manual, which is a part of the KERMIT submission in the CSL/1000. Additional information may also be obtained from the June and July (1984) issues of BYTE magazine, from which I first learned of KERMIT.

KERMIT is a copyrighted protocol developed by Bill Catchings and Frank da Cruz at the Columbia University Center for Computing Activities (CUCCA). They designed and implemented the protocol in order to allow students with PC's to maintain their own archival storage of work done during the semester, to off-load some of the editing-type tasks from the mainframe, and to allow file-transfers for any other reason. Note: the copyright was obtained to protect KERMIT, Columbia University, and the contributors from having the work stolen and then sold as a product. "KERMIT" is a registered trademark of Henson Associates, Inc., creators of "The Muppet Show."

Under this protocol there are two KERMIT programs which mediate the transfer of (usually) text files; one of the programs acts as the sender and the other as the receiver for a given file-transfer operation. In the next file-transfer operation the two programs could swap jobs. It is naturally assumed that each of these programs runs on opposite ends of the same communications link between two possibly dissimilar machines. In addition to transferring files, many KERMIT programs also function as rudimentary terminal-emulators (for the purpose of starting the KERMIT program on the remote system), which eliminates the need to get out of the local KERMIT, then start a separate program to perform this job.

The sender divides the data in a file into more manageable pieces, called "packets." Information is added before and after the data in each packet for control, descriptive, and error-detecting purposes, and then the "data" packet is sent over the communications link to the receiver program. The receiver must determine that the correct packet has been received and that the contents of that packet have been received intact. If there is a problem with the received packet, the receiver tells the sender to try again in a NAK packet, or Negative Acknowledgement. If the packet was properly received, the receiver extracts the data portion of the packet, places it in the file being built, and then informs the sender that the packet was successfully received and that the receiver is ready for more data. It does all of this in an ACK packet, or ACKnowledgement. In addition to sending data packets, the sender also transmits file-name, end-of-file, and end of transmission information in packets. The KERMIT programs can only talk to each other in packets, the general structure of which is shown below and described in the following paragraphs. Note that all packets, regardless of purpose, conform to this structure.

Mark	Len	Seq	Type	Data	Checksum
------	-----	-----	------	------	----------

Each packet begins with the only control-character actually required by the KERMIT protocol, the "mark" character. This is usually a control-A (or SOH) in most implementations, chosen because on most systems the SOH has no special meaning to the communications processor, and so is passed to a program as data. The protocol states that each packet must begin with the packet-marker for synchronization, so if there should occur another marker "within" a packet (probably due to corruption of some part of the previous packet), the previous information is discarded and the "new" mark is treated as if it really is the beginning of a new packet. Both the sending and receiving KERMIT must agree on the choice of the Mark before any packet transfers can occur. Note that no control-characters appear within a packet past the Mark character; all other information within a packet is adjusted such that it is printable! The intent here is to avoid any other control-characters which may have some special meaning to the host system's communications front end, such as a control-D in RTE or control-C in DEC operating-systems. If the mark should be corrupted, the receiver will eventually time-out; the NAK it sends causes the sender to try again.

The second byte of any packet is the Length byte. Because there must be no control-characters within a packet, the length byte is encoded by adding 32 (decimal) to the actual number of bytes within the packet which follow the length byte. [Adding 32 to a "number" in order to make it printable is called the "char" function in KERMIT parlance, which is not to be confused with the FORTRAN character function of the same name.] The length does not include any padding characters which some systems require, nor does it include the end-of-line character required by other systems such as RTE; these characters are considered to be outside of a packet. It does include the sequence-number byte, the type byte, zero or more data byte(s), and one to three checksum bytes; it is the actual packet size minus two. Corruption here will probably cause the checksum to fail, forcing a NAK and a retry.

The third byte of each packet is the sequence number, which ranges from 0 to 64, encoded via char(). If it is corrupted, the receiver gets a sequence or checksum error, to which a NAK is returned, prompting a retry.

The fourth byte of each packet is the type byte, a printable character which tells the receiving KERMIT what kind of packet this is:

- D Data (from a file)
- Y Acknowledge [ACK]; packet may contain special data
- N Negative Acknowledge [NAK]
- S Send Initiate (Send-Init); sets packet parameters
- R Receive Initiate; used to request files from a server
- B Break transmission [EOT]; sent after all file-transfers are done
- F File header; gives the receiver the name of the file to be sent
- Z End of file [EOF]; is sent after each file
- E Error; the data field may contain error text
- G Generic commands. These may not be implemented by all KERMIT programs; a single character in the data field, followed by 0 or more operands, requests host-independent remote execution of a command:
 - L Logout or Bye; terminate KERMIT and log it's session off
 - F Finish; terminate KERMIT but don't log off
 - D Directory query
 - U Disc usage query
 - E Erase or delete a file
 - T Type (list) a file to the terminal
 - Q Query server status

Others as defined by cooperating KERMIT implementations

- C Host command; the data field contains a string to be executed as a system command by the host-system's command processor
- X Text display header; indicates the arrival of text to be displayed on the screen, possibly as a result of a generic or host command. It behaves exactly like a "normal" file-transfer operation, except that the data goes to the screen rather than a disc file.

The various packet types will be examined in more detail below. If the type byte is corrupted, the checksum will be wrong and the receiver will ask for a retry.

The data field, as explained above, may be absent. The contents of the data field are dependent on the packet type: a data packet holds data from a file being transferred, a file-header packet contains the name of the file to be sent, a send-init packet holds parameters pertaining to the pending file-transfer, and the ACK to a send-init packet contains the receiver's parameters for that same file-transfer operation. (The send-init/ACK transaction will be further amplified below.) As before, all data bytes must be printable; non-printable characters are treated in two ways to make them printable depending upon the numeric value of the byte **B** in question:

- a) $0 \leq B < 32$ the control-character b is made printable by XOR-ing it with 64, preceded by a special "quote" character
- b) $B = 127$ treated as in case (a) above
- c) $128 \leq B \leq 255$ All of these character have their "eighth" bit set. The byte is AND-ed with 127, preceded by a "binary" quote character (the "BQuote") if binary-quoting is enabled.

Note that after "binary" quoting, a byte may become subject to "control" quoting, as described in case (a) above. A rudimentary data-compression scheme may also be applied to information in a data packet. To do this, a sequential run of four or more of the same character can be compressed into a 3-byte sequence, (ignoring any possible binary- or control-quoting which may be necessary) by placing a repeat-count character "repc", the count itself [encoded via char()] to make it printable], then the character itself. If the Quote, the Bquote, or the Repc characters

need to be sent, they are preceded by the quote character, but the XOR with 64 is not performed. Not all KERMIT implementations perform binary quoting or repeat-count prefixing. If all eight bits of the RS-232 character are controllable by both ends of the communications link, binary quoting is not even desirable because it could double the transmission-time of half of the (binary) data. All implementations of KERMIT must perform (control) quoting as describe above. A (quoted) CR/LF sequence is used to delimit a logical record within a packet; no prefixing sequences themselves may span a packet boundary, however.

The final field in a packet is from one (default) to three bytes long, and contains the packet's checksum. Each byte is made printable by the char() function described above. All KERMIT programs must implement the 1-byte checksum by simply adding up the numeric value of all bytes between the mark and the checksum itself. Since this sum *s* must be expressed as a single printable character, the 7th and 8th bits of the sum would be lost, so they are extracted and added back in to the sum in the first two bits as:

$\{ s + [(s \text{ and } 300) / 100] \}$ and 77 (in octal arithmetic).

The two-byte checksum is the low-order 12 bits of the same sum, broken into two 6-bit numbers and then made printable via char(). The three-byte checksum is actually a Cyclic Redundancy Check (CRC) formed by the CCITT-recommended polynomial

$$x^{16} + x^{12} + x^5 + 1$$

which is also used in some other networks like DS-1000.

Some machines require a special character to terminate input. If one is needed, the sender appends it after the checksum; it is not considered to be a part of the packet. Similarly, if a machine requires some time to "turn the line around," padding characters are sent as needed; these are also not considered to be a part of the packet.

How KERMITs talk to each other

The following is an example of the sequence of packets only as a file was copied from an HP-1000 "F-series" machine running RTE-6/VM to a VAX 11/780 running under VAX/VMS; none of the operator commands used to cause these events to occur are shown. The information shown is taken from a debug-log of a real KERMIT-RTE session; I have edited only the beginning of each packet to say "RTE" or "VAX" for the sake of clarity, and the lines which are too long have a ".." at the end as EDIT/1000 does in screen mode. The file sent from the RTE ("I/me") to the VAX ("you") was:

```

      This is a short file of test data used to demonstrate how KERMIT
transfers data from one place to another.  It consists of a few very
short
records
to show
how we
can pack multiple records into a single packet, and it contains a ver..
y long record, showing how a record might span a packet.

```

Information shown in (this font) is actual debug-log data; information in (this font) is commentary. The "." is the mark character which must begin each packet.

RTE: ., S~& @~#&1~+

"I want to send you a file." I start the conversation with a 'send-init' packet in which I tell you what special characteristics I expect. Taking the characters, as they appear from left to right,

this is what this packet tells you:

- '.' is the **mark**; it denotes the start of this and all packets
- '.' is the **packet length** in char() encoding; "." = 44, so the packet has 12 bytes following this one (44 - 32 = 12)
- '.' is the **packet sequence-number**, which is also char() encoded. Since "." = 32, this is packet# 0.
- 'S' is the **packet type**. "S" denotes send-init. The data field of this type of packet always has the following format, where fields can be omitted from the right only for each unimplemented feature:

1	2	3	4	5	6	7	8	9	10
Maxl	Time	nPad	PadC	EOL	Qctl	Qbin	ChkT	Repc	Capas...

- '~' is the **Maxl**, or maximum length of packets (in bytes) to send to me, encoded via char(). "~" = 126; maxl = 94. If this field is not supplied (meaning that all send-init parameters were defaulted) then you must assume I will accept 80 bytes.
 - '&' is the **Time** (in seconds) after which you should time me out while waiting for a packet from me, encoded via char(). "&" = 38, so time = 6 seconds, which is reasonable at 9600 baud. RTE KERMIT adjusts this relative to the baud-rate being used! If defaulted, no time-out processing should be performed.
 - '.' is the **nPad**, or number of padding characters I require, using char() encoding. The blank means nPad = 0, so I require no padding. Zero is the default for this field.
 - '@' is the **PadC**, or pad-character to use for padding. The PadC is made printable via the ctl() function, which complements bit-7 of it's argument [ctl(n) = n xor 64]. "@" = 64, so PadC = 0, or "NUL." The NUL would be the default pad-character, but it is ignored if the nPad field is zero.
 - '-' is the **EOL**, or end-of-line character I want at the end of each packet, encoded via char(). "-" = 45, so the EOL should be 15 (CR) which is the default for this field.
 - '#' is the **Qctl**, or the (literal) Quote character to use for control characters within the data field of a data packet. "#" is the default control-quoting character.
 - '&' is the **Qbin**, or the (literal) Quote for binary (8th-bit set) characters appearing in a file's data. Qbin defaults to none: no binary quoting is to be performed.
 - '1' is the **ChkT**, or (literal) Checksum Type to use. "1" signifies a single-byte checksum as described above, "2" says use the 2-byte (simple) checksum, and "3" means use the 3-byte CRC. All KERMITs must at least implement the default 1-byte checksums.
 - '~' is the **Repc**, or (literal) Repeat-count character to use for any data compression, as described above. The default is blank: don't perform repeat-count processing.
- The next field(s) form the **Capability mask(s)**, which are not yet implemented in RTE-KERMIT. These are 6-bit groups [encoded via char()] in which a bit is set to signal a processing capability:

bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
#1	#2	#3	#4	#5	more

The "more" bit is set to indicate the presence of another byte of capability-mask following this one. My KERMIT Protocol Manual (dated 4 November 1983) defines only 3 capability bits:

- #1 Ability to time out
- #2 Ability to accept server commands
- #3 Ability to accept "A" (file-attribute) packets

'+' is the **checksum** of this packet.

If this description seems long, this is a complicated packet.

VAX: ., Yp/ @-#&1~,

The VAX answers, "OK, send it to me using these parameters." This is a normal acknowledge, but an acknowledge to a send-init is special because it contains all of the receiver's parameters in send-init format! Thus this packet says:

',' Y' is the header for a 12-byte packet, packet# 0; "acknowledge."

'p/ @-#&1~' is the packet data for:

Maxl =80 bytes

Time =15 seconds

Npad =0; no padding required

PadC =0 (NUL) is the pad character, ignored since Npad=0

EOL =13 (CR)

Qctl ="#"

Qbin ="&"

ChkT =1-byte checksums to be used

Repc ="~"

No capability masks were sent

',' is the packet's checksum

The exchange of parameters is a short operation; once the receiver has told the sender what it wants, there are no more negotiations. If RTE-KERMIT requests 3-byte checksums and the other end cannot support it, then we must both use 1-byte checksums.

RTE: .+!FTESTFILEU

','+!F' is the header for an 11-byte packet, packet #1, type = file-header. It means, "prepare to receive a file whose name is in the data field." In this case, the file's name is 'TESTFILE' without a type-extension, although type-extensions are supported. The receiver must modify the file's name as needed to conform to the file-naming rules of the receiver's file-system, and to either rename it should there be a conflict with an existing file or to refuse reception, at the receiver's discretion. RTE-KERMIT allows you to overwrite existing files only if you have turned "file-warnings" off.

VAX: .#!Y?

This is an acknowledge to packet #1. If the receiver was unable to accept the file as named for some reason, it could have responded with an error-packet, whose data field would contain the text of an error message stating why the packet couldn't be received.

RTE: .j"D~& This is a short file of test data used to demonstrate how..
KERMIT#M#J9

Since the VAX acknowledged my file-header, I send packet #2 as the first data packet. The "~& " is data-compression for the 7 blanks which begin this record. The "#M#J" are the quoted control-characters "CR" and "LF," which delimit the first record.

VAX: .#"Y@ "OK to packet #2"

RTE: .i#Dtransfers data from one place to another. It consists of a ..
few very#M#

Packet #3 is another data packet. The logical record itself does fit in the packet, but the record-delimiter didn't quite fit.

VAX: .##YA "Packet #3 is OK"

RTE: .i\$D#Jshort#M#Jrecords#M#Jto show#M#Jhow we#M#Jcan pack multiple..
records i]

Packet #4 is also a data packet. It shows how multiple records are packed into a single packet.

VAX: .#\$YB "Packet #4 is OK"

RTE: *i%Dnto a single packet, and it contains a very long record, sho..
wing how aL

Packet #5 is a data packet showing how a logical record might span a packet. In this case, parts of the same logical record span 3 packets.

VAX: *%YC "Packet #5 is OK"

RTE: *C&D record might span a packet.#M#J\

Packet #6 is a data packet holding the last part of that long record.

VAX: *%YD "Packet #6 is OK"

RTE: *#'ZF "Packet #7 is the end-of-file for the "TESTFILE"

VAX: *#'YE "OK on the end-of-file packet, #7"

RTE: *#(B/ Packet #8 is a **Break** packet; "I am done sending."

VAX: *#(YF "OK on packet# 8; good-bye for now"

and the conversation ends. Had there been a problem in any packet transferred to the receiving KERMIT, it would have responded with a NAK packet, and the sending KERMIT would re-send the same packet. If a receiver's ACK packet was corrupted, the sender would, once again, re-send the same packet (the receiver knows to discard duplicate packets - that's why the sequence number is there).

What the user sees

In the following, you will see several complete KERMIT sessions wherein the same group of files (RTE-KERMIT's common files, actually) are transferred from the same RTE-based system to the same VAX. The underlined text is what the user enters in these sessions.

Session 1: the user is already logged on to the VAX system.

\$ kermit

VMS Kermit-32 version 3.1.066

Default terminal for transfers is: _VTA789:

Kermit-32>c backus "C" is short for "connect"

[OSWALD::Connecting to _TXL7:.. Type ^]C to return to VAX/VMS
Kermit-32]

System 2 log on:schumann.54850

PASSWORD ?

SESSION 62 ON 10:37 AM THU., 15 MAY, 1986

PREVIOUS TOTAL SESSION TIME: 75 HRS., 25 MIN., 20 SEC.

This is RTE 6/VM: Rev. A.84 {messages last edited <860515.0929>}

All Backus systems will be DOWN for P.M. on Saturday, May 17

CI.62> wd /kermit (to make getting the files easier)

CI.62> kermit

HP-1000 RTE-KERMIT Version 1.98 <860429.0904>

KERMIT-RTE requires EOL=13!

KERMIT-RTE is in remote-host mode; file transfers are ok

Kermit-RTE> sen .iftn "sen" is minimum command for send; the mask says to get all
FORTRAN include-files in the current working-directory.
^lc (These don't really print - we are returning to the VAX)
[OSWALD::Returning to VAX/VMS Kermit-32]

Kermit-32>rec "rec" means receive; we will take all files sent.
Receiving: KERSTA.IFTN as USR1:[SCHUMANN]KERSTA.IFT;3 [OK]
Receiving: KERCMD.IFTN as USR1:[SCHUMANN]KERCMD.IFT;3 [OK]
Receiving: KERCNF.IFTN as USR1:[SCHUMANN]KERCNF.IFT;3 [OK]
Receiving: KERCOM.IFTN as USR1:[SCHUMANN]KERCOM.IFT;3 [OK]
Receiving: KERDBG.IFTN as USR1:[SCHUMANN]KERDBG.IFT;3 [OK]
Receiving: KERFIL.IFTN as USR1:[SCHUMANN]KERFIL.IFT;3 [OK]
Receiving: KCMNDS.IFTN as USR1:[SCHUMANN]KCMNDS.IFT;3 [OK]
Kermit-32>c

Once RTE-KERMIT has sent all files given by the mask, it sends the "break" packet to the
VAX, which allows it to accept more user commands. We re-Connect to shut off the
RTE-KERMIT.

[OSWALD::Connecting to _TXL7:. Type ^]C to return to VAX/VMS
Kermit-32]
Kermit-RTE> ex
CI.62> ex
Finished
CI.62 REMOVED

SESSION 62 OFF 10:39 AM THU., 15 MAY, 1986
CONNECT TIME: 00 HRS., 02 MIN., 35 SEC.
CPU USAGE: 00 HRS., 00 MIN., 15 SEC., 140 MS.
CUMULATIVE CONNECT TIME: 75 HRS., 27 MIN., 55 SEC.
END OF SESSION

^lc (Once again, not printed; back to the VAX)
[OSWALD::Returning to VAX/VMS Kermit-32]
Kermit-32>ex We're done with the VAX, so we log off.
\$ lo
SCHUMANN logged out at 15-MAY-1986 10:40:30.35

Session 2: The user is directly connected to the RTE-based system

System 2 log on:schumann.54850
PASSWORD ?

SESSION 69 ON 10:42 AM THU., 15 MAY, 1986
PREVIOUS TOTAL SESSION TIME: 75 HRS., 27 MIN., 55 SEC.

This is RTE 6/VM: Rev. A.84 {messages last edited <860515.0929>}

All Backus systems will be DOWN for P.M. on Saturday, May 17

CI.69> wd /kermit
CI.69> kermit
HP-1000 RTE-KERMIT Version 1.98 <860429.0904>

KERMIT-RTE requires EOL=13!

```

KERMIT-RTE is in remote-host mode; file transfers are ok
Kermit-RTE> set L 62      "L" is short for "line"; LU 62 is the direct VAX link
KERMIT-RTE is in local-host mode to LU 62 @ 9600 baud; Parity = NONE
  As KERMIT-RTE begins to use another LU for file-transfers, it identifies that it really is on a
  configured 12040 or 12792 MUX port, then shows you the baud-rate and parity of the line.
Kermit-RTE> c              "c" is short for connect (through LU 62)
[connecting to LU 62; return via "control-]" then "C"]

```

E-Systems Engineering Computer Center VAX

Username: SCHUMANN

Password:

```

      Welcome to VAX/VMS version V4.2 on node OSWALD
      Last interactive login on Thursday, 15-MAY-1986 10:33
ENGINEERING VAX
NODE: OSWALD
LAST BOOTED: 8-MAY-1986 19:06:46.08

```

\$ kermit

VMS Kermit-32 version 3.1.066

Default terminal for transfers is: TXL7:

```

Kermit-32>rec              Once again, the VAX takes all files we send
                          (These don't print, but we return to RTE-KERMIT)
^lc

```

[back at KERMIT-RTE]

Kermit-RTE> sen .iftn

As before, send all files in mask "@.iftn" The numbers below are a running packet count; the file-header for the first file (KERSTA.IFTN) was packet #2, the header for KERCMD.IFTN was #12, etc. The numbers after the "/" give counts for retries; none occurred in this session.

```

1/000 Sending KERSTA.IFTN:::4:3:38
11/000 Sending KERCMD.IFTN:::4:3:37
20/000 Sending KERCNF.IFTN:::4:4:37
35/000 Sending KERCOM.IFTN:::4:10:38
67/000 Sending KERDBG.IFTN:::4:3:37
76/000 Sending KERFIL.IFTN:::4:6:38
93/000 Sending KCMNDS.IFTN:::4:4:36
106/000 File transfer completed

```

```

Kermit-RTE> c              Re-Connect in order to log off the VAX
[connecting to LU 62; return via "control-]" then "C"]

```

Kermit-32>ex

```

$ lo
SCHUMANN          logged out at 15-MAY-1986 10:47:57.52

```

```

^lc              (Escape back to local: RTE-KERMIT)

```

[back at KERMIT-RTE]

Kermit-RTE> ex

CI.69> ex

Finished

CI.69 REMOVED

```

SESSION      69  OFF 10:48 AM  THU., 15  MAY , 1986
CONNECT TIME:                00 HRS.,   05 MIN.,   31 SEC.
CPU USAGE:                   00 HRS.,   00 MIN.,   20 SEC.,   10 MS.
CUMULATIVE CONNECT TIME:    75 HRS.,   33 MIN.,   26 SEC.
END OF SESSION

```

Using KERMIT terminal-emulators can be a real problem; this is especially true of the RTE-version

of KERMIT: you must type very slowly, and not all of your typing will be echoed back. There is an alternative called the SERVER. Once invoked, the server takes it's commands from packets generated by the local KERMIT.

Session 3: you are already logged on to the VAX and will invoke the RTE-KERMIT server.

```
$ kermit
```

```
VMS Kermit-32 version 3.1.066
```

```
Default terminal for transfers is: _VTA789:
```

```
Kermit-32>c backus2 (As before, Connect to the correct RTE system)
```

```
[OSWALD::Connecting to _TXL7:: Type ^]C to return to VAX/VMS  
Kermit-32]
```

[log on as usual]

```
CI.62> kermit ser KERMIT-RTE allows one command in the run-string
```

```
HP-1000 RTE-KERMIT Version 1.98 <860429.0904>
```

KERMIT-RTE requires EOL=13!

KERMIT-RTE is in remote-host mode; file transfers are ok

[KERMIT Server running on an HP-1000 host.

You must escape to your local machine now!]

```
^]c Escape back to VAX KERMIT
```

```
[OSWALD::Returning to VAX/VMS Kermit-32]
```

```
Kermit-32>get /kermit/.iftn
```

Note the use of the "get" command (not receive) and a full path-name for the files to be received, assuming that I didn't already set the working-directory to /KERMIT. A "get" command causes the local KERMIT to generate a "receive-init" (type = R) packet whose data is the filespec given as the command's parameter. The server's response to this type of packet is a send-init rather than the usual ACK packet. Some KERMITs will allow a second parameter which could rename received files.

```
Receiving: KERSTA.IFTN as USR1:[SCHUMANN]KERSTA.IFT;4 [OK]
```

```
Receiving: KERCMD.IFTN as USR1:[SCHUMANN]KERCMD.IFT;4 [OK]
```

```
Receiving: ... (etc.)
```

```
Receiving: KERFIL.IFTN as USR1:[SCHUMANN]KERFIL.IFT;4 [OK]
```

```
Receiving: KCMNDS.IFTN as USR1:[SCHUMANN]KCMNDS.IFT;4 [OK]
```

```
Kermit-32>f "f" is short for finish, which terminates the remote KERMIT, but  
doesn't log the session off.
```

The user would re-connect to the remote session [not KERMIT] to log off, then escape back to the local KERMIT (which is providing the terminal emulation) to do other processing or to log off.

The example above illustrates a very important point. The connection between these two "mainframe" computers is not a dial-up, but a permanent, hard-wired connection. Under these circumstances, it is a bad idea to use the Logoff or BYE commands with a server because the messages associated with the logging-off operation may be interpreted as attempts to log on to the systems. Since these messages probably will not be valid user/password sequences, the "log-on failure" message will be generated, and that might be interpreted by the other system as a log-on attempt, etc. I have personally seen some (DEC) systems which were brought to their knees by this type of "vicious cycle" unleashed by an unknowing user. The "most correct" method for logging off a hard-wired connection is that shown above: Finish, then Connect, log off manually through the local KERMIT's terminal-emulator, then escape back to the local KERMIT.

Session 4: you are directly connected to the RTE-based system and will interact with the VAX-KERMIT server.

```
CI.69> wd /kermit
CI.69> kermit
HP-1000 RTE-KERMIT Version 1.98 <860429.0904>
```

KERMIT-RTE requires EOL=13!

```
KERMIT-RTE is in remote-host mode; file transfers are ok
Kermit-RTE> c 62      (RTE-KERMIT can set line and connect in one step!)
KERMIT-RTE is in local-host mode to LU 62 @ 9600 baud;  Parity = NONE
[connecting to LU 62; return via "control-]" then "C"]
```

<< The usual log-on stuff >>

```
$ Kermit ser          (VAX-KERMIT will take run-string commands)
VMS Kermit-32 version 3.1.066
Default terminal for transfers is: _TXL7:
```

[Kermit Server running on VAX/VMS host. Please type your escape sequence to return to your local machine. Shut down the server by typing the Kermit BYE or FINISH command at your local machine.]

^lc escape back to RTE-KERMIT

[back at KERMIT-RTE]

```
Kermit-RTE> sen .iftn
```

```
1/000 Sending KERSTA.IFTN:::4:3:38
```

```
11/000 Sending KERCMD.IFTN:::4:3:37
```

```
...
```

```
93/000 Sending KCMNDS.IFTN:::4:4:36
```

```
106/000 File transfer completed
```

(You could send another group of files here, if desired, but we're done for now.)

```
Kermit-RTE> f      Terminate remote KERMIT, but don't log off.
```

```
Kermit-RTE> c      Re-connect in order to log off safely.
```

Note that sending to a KERMIT acting as a server doesn't look very much different than sending to an "interactive" KERMIT. The advantage is that you need not type slowly or worry that the terminal-emulation provided by the local KERMIT has dropped or failed to display a key stroke, nor are you required to keep switching back and forth between the local and remote KERMIT programs.

Section 2: KERMIT vs. other protocols

In this section I will briefly describe some other file-transfer methods, the XMODEM, PCLINK, AdvanceLink™, and CompuServe "B" protocols, and compare them to the KERMIT protocol. While this paper is about KERMIT, I will try to be objective.

XMODEM Protocol

XMODEM is probably the best-known of file-transfer protocols. It was developed by Ward Christenson for the purpose of reliably mediating the transfers of files for the users of a CP/M-based bulletin-board system. My information on it comes from a document called "MODEM Protocol Overview," uploaded to CompuServe by Ward Christenson himself, dated January 1, 1982.

Like KERMIT, XMODEM is an asynchronous, half-duplex protocol which divides the data into packets of convenient size whose bounds have no particular correspondence to the logical records within them. Unlike KERMIT, **only file data is transmitted in packets**. An XMODEM packet has the following structure:

SOH	BLK#	$\overline{\text{BLK\#}}$	128 Data Bytes	Hi Chksum	Lo Chksum
-----	------	---------------------------	----------------	-----------	-----------

Each packet begins with the control-A (SOH) character, like KERMIT. The second byte of each packet is the block-number. It starts at 1, incrementing once for each packet successfully sent, and wraps from 255 to 0 (not 1, as you might expect). The third byte of a packet is the one's complement of the same block# given by the second byte. The 4th through 131st bytes of a packet contain file data, with each logical record terminated by a (CR) and (LF), as in KERMIT. XMODEM does not allow "short" packets; the last packet must be padded to 128 data bytes even if it only contains the final (LF) for the last record. There is no requirement that all the data be printable. The final two bytes of a packet contain the 16-bit simple checksum of the data bytes only, ignoring any carries. There are some extensions to this protocol which compute the same CCITT-CRC as is optional in KERMIT. The XMODEM standard requires 8-bit characters with no parity; there are no specifications for a 7-bit plus parity implementation. An ASCII-only or unpacked-hex method could be arranged easily enough (if both sides agree) by **AND**-ing the non-data portions of the packets with 127 decimal, for those machines which have no control over the parity of their communications ports.

The most major difference between the KERMIT and XMODEM protocols (aside from differences in packet structure and length) is the method in which the receiving program performs handshaking with the sender. KERMIT puts all "answers" into packets; XMODEM uses a single character to acknowledge (05H - ASCII "ENQ"), negative-acknowledge (15H - ASCII "NAK"), or report errors (18H - ASCII "CAN") in received packets. A difference in the sender's operation is that it waits for the first timeout from the receiver before it even sends the first packet; at end-of-file it sends only a single character (04H - ASCII "EOT") to indicate that the sending of the file is complete rather than a complete packet, as KERMIT does.

The result of these is that XMODEM has the potential to be faster than KERMIT:

- a) A longer data packet, without special prefix characters, has a lower overhead (more data bytes per byte of control)
- b) A shorter response handshake sequence also cuts overhead

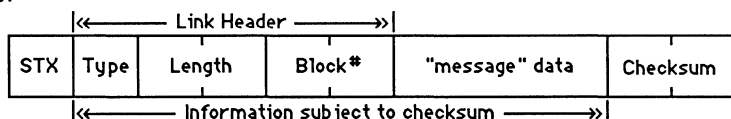
On a good communications line (non-modem?) this potential can be realized. On a poor line, the longer packet length will be subject to corruption more often; with KERMIT you can reduce the packet size and perhaps transfer a file on a line which could be too noisy for the XMODEM protocol. Since XMODEM has only one "type" of packet (the data packet), there are no provisions for a file-server or other extensions of which KERMIT is capable.

PCLINK Protocol

PCLINK is a file-transfer protocol developed by Walker, Richter, and Quinn to be used between HP 3000 or VAX machines and a terminal-emulation program for IBM PC's (and clones) originally called PC2622; it is now called REFLECTION™. DIAS Corporation has also implemented the protocol for HP 1000's. My information on this protocol comes from a document file called "Protocol.doc," and from the SPL/3000 source to PCLINK, both of which used to be supplied with copies of PC2622.

While it is currently in somewhat limited use, folks at INTEREX tell me that this protocol is becoming more and more popular; PCLINK does deserve some examination because it combines some of the features of both KERMIT and XMODEM. Like XMODEM, it utilizes a single byte to report success or failure of the last packet received, and it has a method for simultaneously acknowledging the last packet received and then "turning the line around." Like KERMIT, it has several packet types and a semi-variable (but larger) packet size; the "master" (the program running on the PC) normally "tells" the slave (running on the HP-1000/3000 or VAX) to send or receive a file like a KERMIT server.

PCLINK packets are normally 128 bytes (unless changed by the master) and have the following structure:



- All packets begin with the "STX" (decimal 2)
- The type byte is one of
 - 0 (ASCII "NUL") to change the standard block size; the minimum packet size is 128 bytes; the SPL/3000 program has a maximum packet size of 524 bytes, of which 512 are data.
 - 1 (ASCII "SOH") for a data packet
 - 3 (ASCII "ETX") for a dummy packet, used for synchronization purposes
- The Length bytes show the size (in bytes) of the data portion
- The Block# bytes begin at zero, incrementing for each packet. A block# of zero indicates that the packet should be accepted even if it is out of sequence.
- The "message" portion of a packet is variable length; in a type-0 packet (change the standard block size) two bytes give the new block size as a 16-bit number. All type-1 (data) packets begin with a 16-bit "message-type," possibly followed by more information or file data. Message types include:
 - 0 Initial message; gives the slave's software version number
 - 3 Start message; gives the name, record size, file size, type of transfer (ASCII or Binary) and direction of transfer (to or from the master)
 - 4 Answer start message (a response to the start message?); contains an error code (0 means no error), error text, the record length, and file size of the slave's file
 - 5 Data message; this is the type of packet which transfers file data. I don't know if there is any kind of quoting mechanism within the data, or if such is needed. PCLINK does perform some data compression.
 - 6 End of data; contains no data and apparently signals the end of the file, like KERMIT's "Z" packet
 - 7 Slave aborting; signals the master that for the reason given by the error code and text in this packet, the slave has died
 - 8 Answer end of data; reports errors when the slave cannot close the file it received
 - 9 Terminate slave; the master tells the slave to terminate
- The checksum is computed using the same CCITT-CRC polynomial used for KERMIT's optional 3-byte checksum.

Unlike KERMIT, the PC running the PC2622/Reflection™ program is always the "master" regardless of the direction of file transfer. The handshake is a single character, like XMODEM, where "S" means success (ACK), "F" means failure (NAK), and "T" means "success and turn the line around" (another kind of ACK). The "S" and "T" handshakes are unusual; the "S" seems to be for the benefit of the file-recipient to allow it to say, "I got the last packet OK, but I need to say

something now," whereas the "T" handshake says, "The last packet was ok; send some more."

PCLINK and XMODEM share some of the same strengths and weaknesses, and for the same reasons: packet size and brevity of the handshake. PCLINK has the potential for even better throughput than XMODEM due to the larger amount of data a packet can carry, and better data reliability due to the CRC, but only on a relatively noise-free line. It would probably be frivolous to try to use the maximum packet size on most modem connections because at 1200 baud, a packet would take almost three and one-half seconds to transfer; at 300 baud it would take almost fourteen seconds! It appears that the PCLINK protocol does support file masking and the sending of several files with one command.

CompuServe "B" Protocol

I was fortunate to be able to bring this to you: as I was nearing the completion of this paper the June, 1986 issue of *Doctor Dobb's Journal of Software Tools* arrived containing a brief description of the protocol and most of a program which implements that protocol. The article is entitled "The CompuServe B Protocol: A Better Way to Send Files," by Levi Thomas and Nick Turner; in it the authors say that this is a better way to transfer files than XMODEM due chiefly to the fact that XMODEM will time out if there are any delays in CompuServe's processing.

The CompuServe B protocol is invoked by the user, who requests a file upload or download. Once invoked, the host (CompuServe) becomes the master and the PC becomes the slave, in a reversal of PCLINK's method. The master and slave exchange some preliminary information about the slave's capabilities, then the file-transfer begins in (mostly) fixed-length packets of 516 bytes:

DLE	- TYPE	SEQ #	DATA	CHECK
-----	--------	-------	------	-------

- Each packet begins with the ASCII character "DLE" (Decimal 16)
- A data packet's type is "B"
- The sequence number is next; this is a literal "0" through "9" and wraps back to "0"
- The data field is next; the first byte of the data is another type code, the next 510 bytes are data from the file being sent or received, and the last byte is the ASCII "ETX" (decimal 3) character. If any of the data bytes are one of the B protocol's special characters (ASCII NUL, ETX, ENQ, DLE, NAK, XON, or XOFF), they are prefixed by a DLE, and then the special character is modified by adding the "@" character (decimal 64) to it.
- The check character is built from the data field only by:
 - 1) shifting the previous checksum left one bit
 - 2) If the checksum now exceeds 8 bits, it is masked back to 8 bits and 1 is added to it to account for the "carry"
 - 3) A data character is added to the checksum
 - 4) The checksum is adjusted again as in step 2
 - 5) The checksum is then placed into the packet as a single character.

The last data read from a file may not completely fill a packet, so the last packet of an upload or download could be short. The B protocol then specifies that a "transfer packet" be sent to close the file; this is a data packet formed like the above, but the first character is "T" for transfer, and then "C" (presumably for close). The receiver of the file may suspend the transmission by sending an ASCII XOFF (decimal 19, control-S) and resume it with an ASCII XON (decimal 17, or control-Q). If the receive has no problems with a given packet, it sends a DLE followed by the

sequence number of the packet received. If there was a problem with the packet, the file's receiver sends a single ASCII NAK (decimal 21) character. The slave can abort the transfer in progress at any time by sending a data packet with an "A" (for abort) or "E" (for error) as the first data byte, and text following that character for explanation, if desired. A packet may be retried up to ten times; a packet will be NAK-ed if the receiver must wait more than ten seconds for a byte of any packet, or if an XOFF condition remains on the line for more than ten seconds.

This protocol bears a fair resemblance to KERMIT. Like KERMIT, special characters must be quoted, but there is no requirement that all information in a packet be printable as in KERMIT. The data portions of a packet are marked with a special start and end character, and since most of the packets containing file data are fixed at 516 bytes, there is no need for a length field. It seems odd that a NAK operation only requires a single character which is not associated (by sequence number) with the packet being refused, but an ACK operation does associate with the received packet. The checksumming method is probably as effective as KERMIT's 1-byte checksum. This protocol is certainly not one to be used on a noisy line!

AdvanceLink Protocol

I have attempted to obtain a description of the AdvanceLink protocol, but it did not arrive in time for this paper. Current indications are that this protocol is considered "too sensitive" to reveal to the HP user public in this forum. I am therefore resorting to two second-hand sources of information on the performance of this protocol, compared to the others described above.

In his article entitled "PCs and HP 1000s: No Longer Strange Bedfellows" (TC Interface, Volume 5, Issue 3, the May/June 1986 issue), Loyd Case (Jr.) compared KERMIT, XMODEM and AdvanceLink protocols. While there was a passing reference to PCLINK protocol in his article, he never quite got around to talking very much about it. His appraisal of the other three protocols was that the best performance could be obtained from AdvanceLink, followed by XMODEM, then KERMIT. He noted that AdvanceLink only works between HP's PCs, IBM PCs (and clones?), and HP 1000s (and 3000s) running the Monitor program. He felt that if other PCs, 1000s, and/or DEC machines needed to talk, that XMODEM was the protocol of choice. I don't know why he had so little to say about KERMIT; evidently he doesn't transfer many files between 1000s, between a 1000 and a 3000, or between either of those and many DEC machines.

For another point of view, we turn to Jack Armstrong, writing for The Chronicle (Volume 3, Number 7a, the June 1986 issue), in an article called "Is There a Cure for Terminal Emulation?" He compared PCLINK to AdvanceLink, running from an HP 150 and a Vectra to an HP 3000. He admitted greater experience with the PC2622/Reflection™ software than with AdvanceLink 2392, but he claimed to be unbiased in his comparison. He complained at the lack of real information in the AdvanceLink documentation, but that it was "very readable." In terms of performance in transferring files, PCLINK came out the clear winner with a reported 20% to 30% difference in transfer time. The only concession Mr. Armstrong made to AdvanceLink was that it could be better than the PCLINK protocol in the future only because of its interface into AdvanceNet, HP's IEEE 802.3 Local Area Network product.

Section 3: A KERMIT User's Manual

General Information and Warnings

In writing this implementation of KERMIT, I tried to make it as much like other versions as possible,

to shorten the learning cycle for those already familiar with some other KERMIT implementation. The most sophisticated versions I have found are for VAX (under VMS) and PDP-11 (under RSX) machines; these versions are the model for the RTE version I contributed to the CSL.

There are a few items of which the user should be aware before running KERMIT. KERMIT-RTE is not the "friendliest" program in the world. I have tried to put as much helpful information into KERMIT's help file as possible, but there are some topics which would only be of interest to a KERMIT implementer not the user. In the commentary to the help file below, I may refer to the protocol description at the front of this paper or to the protocol manual; this information is probably not necessary knowledge for the average user. There are some things (mostly external to KERMIT-RTE) which all users should know; these things could affect the health of the systems used on either end of any KERMIT link!

Warning #1: KERMIT-RTE modifies the configuration of the port(s) it uses on a 12040B/C (A-series) or 12792B/C (M/E/F-series) multiplexer. When KERMIT terminates normally, it restores any parameters it has changed. For this reason, NEVER ABORT KERMIT! KERMIT-RTE does check the break-flag inside some of it's processing so that if a user has lost control, KERMIT can be made to either return to command mode or to terminate, depending on what it is doing at the time. If you abort KERMIT-RTE, you will probably have to re-issue the control requests used at bootup to set up the affected port(s), as well as restoring the timeout value to some reasonable value. It should be noted that sometimes the affected ports may not be restored until the system is re-booted. KERMIT-RTE can modify timeout, handshake, type-ahead, and parity.

Warning #2: Never use the "bye" command (to a server) if the two mainframe systems (defined as a system which requires you to Log-on) are connected via a permanent hard-wired connection. If you should do this, you may discover that both systems will be trying to log each other on, using the end-of-session and failure to log-on messages generated by the opposite end as user names and/or passwords, which, of course, are doomed to fail. KERMIT-RTE tries to avoid this by "sleeping" for 5 seconds when you terminate a two-port session, but that may not always be sufficient.

Caution #1: KERMIT-RTE has been known to "tie up" a port which it did not use during an otherwise "normal" session. The condition is recognized by the port echoing back any characters it receives, including three or more carriage returns, and the affected port is always on the same mux card that KERMIT did use. The problem is intermittent and seems to be a mux firmware or hardware problem. You can usually fix the situation by starting, then aborting any kind of read operation (like "L") against the affected port(s). It is usually not necessary to reconfigure a port in this condition, and on occasion, reconfiguring these ports may render them unusable until the system is re-booted. Note: if a port stops responding after two carriage returns, that port is probably in type-ahead rather than the condition I have just described.

Caution #2: If you should Connect to an LU on which KERMIT is commonly run, and you see "# N3", Don't Panic! What you have just seen is a NAK packet (using 1-byte checksums), which means that the other end is probably running KERMIT either as a server or actively trying to transfer a file. If you escape back to the local KERMIT and give almost any server command (like Finish) you will probably regain control of the other end of the connection. This is true for most versions of KERMIT you may run, not just the RTE versions.

Caution #3: The KERMIT-RTE should not be used on an A-series machine if it will be used with a noisy (modem?) line. Under RTE-A, because of a problem in the MUX device driver, KERMIT

bypasses the device driver and so it loses the ability to timeout a read request. It is the ability to timeout which allows any version of KERMIT to deal with noise-induced corruption in a packet. When you hear about a revision 2.00 or later of the RTE-KERMIT, this problem will have been solved, because the driver will have been fixed!

Caution #4: KERMIT-RTE can only transfer files to another machine connected via the MUX cards noted above. You may, however, control KERMIT-RTE from a terminal which is connected to the 1000-series machine on any type of interface card. The only problems you may incur under these circumstances would be loss of data while in the terminal-emulation mode, especially if you are connected via a 12531 card on an M-, E-, or F-series machine.

RUNNING KERMIT

Just enter "KERMIT." If you would like, you may enter one KERMIT command in the run-string. If it is not SERVER, EXIT, or QUIT, then once the command is completed KERMIT-RTE will request more commands from your terminal. If the command entered in the run-string is "TR <file-name>," KERMIT-RTE will completely execute the transfer-file named before returning to the terminal for more commands (assuming that the transfer-file doesn't contain a SERVER, EXIT, QUIT, or BYE command). In order to be able to get on-line help, KERMIT.HLP must be in your working-directory (if active), in the SYSTEM directory, or in the KERMIT directory, or it must be named "KERMI in FMGR space. KERMIT.HLP (or "KERMI) must be a type-1 file prepared with GENIX (which is why there is a GENIX.RUN in the CSL submission - for RTE-A users).

In the annotated version of the help file (supplied with the RTE versions of KERMIT) which follows, unless noted to the contrary, the information here applies equally to the RTE-A and RTE-6 versions of KERMIT. Information which appears in (this font) is text from the help file, (this font) will give the additional information or explanations which don't conveniently fit in the help file. The help-file text is always shown first. As usual, parameters appearing in square brackets ("[" and "]") are optional parameters. In the following there will be references to the "local" and the "remote" KERMIT. The local KERMIT (RTE version or otherwise) is the one logically "closest" to your terminal; if you are running a PC KERMIT on your end of the communications line and RTE-KERMIT on the other end, your PC is the local KERMIT and KERMIT-RTE is the remote one. If, on the other hand, you are running KERMIT-RTE (either through a terminal emulation on a PC or on a terminal) which is connected to another machine on some LU other than (session-LU) 1, your PC or terminal is talking to the local KERMIT, and the machine on the other LU is the remote one.

GETTING ON-LINE HELP

This info goes with KERMIT version 1.98 or later, as of June 2, 1986.

KERMIT is a file-transfer protocol for use over an asynchronous serial telecommunications lines. Files are broken up into "packets", adding checksums and other control information to ensure, with high likelihood, error-free and complete transmission.

KERMIT-RTE is implemented for HP-1000 systems running RTE-6/VM or RTE-A. The following commands may be entered in upper- or lower-case:

Bye	Connect	Exit	Finish	Get	Help	Quit	REceive
RUn	SET	SEnd	SErver	SHow	STatus	Transfer	

Note: the minimum allowable abbreviation is shown in UPPER-CASE.

Info about a given command's parameters, if any, is available via a "?" "SET ?" gives you a list of settable parameters. Info about the commands themselves is available via "HELP <command>" where you replace "<command>" with one of the commands shown above. If you need even more help, consult the KERMIT user's manual.

On the first line of general help information you will see the revision-code and date of the help file. If this differs from the version and date of the KERMIT-RTE you are using (displayed as you first run KERMIT), you may not be able to get help on all commands. I will always try to maintain compatibility between versions of KERMIT-RTE as I improve it, but this is not always possible. For instance, the 1.95 version was the first one I released to the CSL at the Washington conference in 1985; it contains commands which are no longer present because I either didn't think they were needed that much, or because I needed the space.

KERMIT-RTE has a user interface which will seem unusual:

- It does accept lower-case commands
- It doesn't have a command-stack
- It will give you very brief assistance if you enter a "?"
- It allows you to abbreviate commands, some as short as 1 character
- It requires you to spell correctly if you decide to spell out a command

The reasons for this are that KERMIT cannot assume it is going to be interacting with a terminal made by Hewlett-Packard, and this user interface is similar to the PDP-11 KERMIT user interface. If you always use the first three characters of a command, you will always be correct. If you use one or two characters and there are two or more commands which start with those, KERMIT will remind you that what you have entered is ambiguous and matches those commands.

Most of the time the information you need about a given command can be obtained by entering "Help, <command>." It is possible to know all you need for "normal" file-transfer operations without ever opening a user's or protocol manual. I will try to at least direct you to the sources of information beyond the help file in these comments.

TERMINATING KERMIT

EXIT or QUIT causes this KERMIT to shut itself down in an orderly way (as opposed to crashing!), closing debugging files if opened, and restoring various things to their original state before KERMIT was run.

If you are running KERMIT-RTE as a server, you must use the FINISH, BYE, or LOGOFF commands from your local machine rather than QUIT or EXIT.

BYE or FINISH

FINISH causes the remote KERMIT to terminate but not log-off from the remote system when the remote KERMIT is acting as a server. It does not cause the local KERMIT to terminate.

BYE causes the remote KERMIT to terminate AND log-off from the remote system when the remote KERMIT is acting as a server. It DOES cause KERMIT-RTE to terminate as if an EXIT or QUIT command was given.

WARNING: You should NOT use the BYE command under any KERMIT if there is a permanent connection between the systems, and if some kind of

"log-on" is usually performed on BOTH systems. If you do use the BYE command in such a situation, one system's log-off messages could act like a log-on attempt to the other system, and since that is bound to fail, the resulting message will be like a log-on attempt to the first system, resulting in a chain-reaction which has been known to seriously degrade system performance.

As stated before, you should never abort KERMIT-RTE! KERMIT changes the configuration of the MUX port(s) through which it transfers files, but it will only restore that configuration if it terminates normally; you won't be able to "fool" KERMIT by restarting it and then terminating.

If KERMIT-RTE is running as a (remote) server, you must use the FINISH, BYE, or LOGOFF commands (as available in the local KERMIT version) to terminate the server. If KERMIT-RTE is talking to a server, you may use the BYE command to terminate both the local KERMIT-RTE and the remote KERMIT simultaneously, assuming that the remote KERMIT's server supports this. You should never use a BYE or LOGOFF command if the connection is between two mainframe machines with a permanent hard-wired connection!

RETRIEVING FILES FROM ANOTHER SYSTEM

```
GET <file-descriptor> [<receive file-mask>]
RECEIVE [<receive file-mask>]
```

GET and RECEIVE tell this KERMIT to receive one or more files from the KERMIT running on the other computer, be it a PC or another mainframe. If the other KERMIT is a SERVER, you MUST use GET instead of RECEIVE. GET requires a file-descriptor which must be legal for the remote system, and may contain wild-cards if the remote KERMIT accepts them.

The optional <file-mask> parameter allows you to rename all received files according to that mask; if a full file-name and path is given, the first file received will use that full name, and all remaining files in the group will use all but the file-name part of the mask. If the received files are put in FMGR space, their names will be subject to (possibly severe) editing, since FMGR allows only 6 characters. FILE-WARNING NOTE: If a received file's name conflicts with an existing file, the RECEIVE or GET will be aborted with a message unless file-warnings are off. If file-warnings are off, new files will overlay the first existing one with the same name. Do "HELP SET WARNING" if you need more information on this.

Files are moved from machine "A" to machine "B" using one of two methods:

- The 'SEND-RECEIVE' method requires you to interact with both of the machines doing the transfer, and to do that you must do all of the work for one of the machines through the other KERMIT's terminal-emulator. Examples of this method are the first two sessions shown under "What the User Sees," above.
- The 'SERVER' method only requires that you start the server, and then you can give all of the commands needed from the local KERMIT. Examples of this method are the third and fourth sessions shown under "What the User Sees," above.

Received files are normally named using the name of the file on the "other" system. If this is undesirable, you may use a file mask as an optional parameter on either the GET or RECEIVE commands to rename the received file(s). The first file received will use all parts of the file mask in

it's name; any subsequent files received in the same group will use all but the file-name part of the mask, if given. In this manner, you can change only the type-extension of received files, if desired. Files will be put in the user's working-directory, if active, or in FMGR space if there is no working-directory; this can be overridden if the optional file mask includes a file path. As noted above, if a file's destination is a FMGR cartridge, the name used on reception will be truncated to the first six characters if the name is longer than that.

If the name of a file to be received (after the optional renaming) conflicts with an existing file in the same file path (or first mounted FMGR cartridge), KERMIT will normally abort the transfer rather than overlay the file. If you really want to overlay the file, you should SET WARNINGS OFF. Note that as a file is overlaid, it is not purged and then recreated; an end-of-file is written as the first record, then the file is completely rewritten.

TRANSMITTING FILES TO ANOTHER SYSTEM

SEND <file-descriptor> [<first file-name>]

"SEND <file-descriptor>" causes KERMIT-RTE to send the file(s) matching the file-descriptor to the other KERMIT whether or not it is a SERVER.

"SEND <file-descriptor> <first file-name>" works as above starting with the given file-name (wild-cards are NOT allowed here!); this is used primarily to resume sending a set of files after some kind of interruption.

Files' names are sent to the "other" KERMIT in a "packet" so that they will know what to receive. If the SEND command is issued to the remote KERMIT (you are CONNECTed) then you must escape back to the local KERMIT to give a RECEIVE command within 15 seconds or this packet may be lost. If this is not an appropriate delay, you may alter it using the SET DELAY command.

Files are sent to a remote KERMIT or KERMIT server using the SEND command as described above. If you want to tell the remote KERMIT to send a file to the local one, you must CONNECT to the remote KERMIT, use the SEND command, then escape back to the local KERMIT and give the RECEIVE command, all within the remote KERMIT's delay time. KERMIT-RTE's delay time is 15 seconds by default and is changed via the SET DELAY command. Other KERMITs' delay times may be different, so you should check.

If you need to transfer a group of ten files, and for whatever reason the transfer is aborted at the eighth one, you don't necessarily need to purge the seven files successfully received and then re-send the whole group, nor must you manually send the remaining three. (You may need to purge the file whose transfer was aborted, however.) You can use the same SEND command, with the same file-descriptor (wild-card characters are OK), and with the eighth file's name as the second parameter in the SEND command, and KERMIT-RTE will locate that file in the group, send it, and send the remaining two files in the group. KERMIT tries to match the "first file name" on a character-by-character basis to each file in the group until it matches, so you need not specify the entire name if the first few characters will match only the desired file in the group. Once the desired file has been transferred, KERMIT-RTE will resume sending the files that would have normally followed it anyway. The file-masking routines will always return the names matching the mask in the same order unless there is another file created matching the mask between the times the two masked searches are performed.

MAKING A CONNECTION TO ANOTHER MACHINE

CONNECT [<lu#>]

The CONNECT command puts KERMIT into a terminal-emulator state, connecting your terminal to either the <lu#> in this command, if given, or to the LU# given in a previous SET LINE command. Anything you type will be sent to the other "remote" computer; anything it sends back will be displayed on your terminal. WARNING: HP-1000 systems currently do not adapt well to this sort of thing, so be patient and TYPE SLOWLY!

When you are typing to KERMIT-RTE, it will prompt you with "KERMIT-RTE>" unless you have changed the prompt (see SET PROMPT for info). When you CONNECT to another machine, you will see information about how to ESCAPE back to KERMIT-RTE, and then you will be typing as if you were on a terminal directly connected to the other machine. When you need to return to KERMIT-RTE (to give it a command or to exit), you will need to type the ESCAPE info as shown in the message you got when you did the CONNECT (see SET ESCAPE for more info). You will then get a message about having returned to the local machine and KERMIT-RTE.

In addition to being able to connect your PC with an HP-1000 system and transfer files, you can also connect your PC or terminal "through" the 1000 system and transfer files between the 1000 and another mainframe having a KERMIT program. The first step in this process is to connect to that other machine and log on. The CONNECT command allows this to happen by acting as a terminal-emulator between your terminal or PC and the other "remote" system. Before you can successfully connect, you must either specify an LU as a parameter to this command, or you can use the SET LINE command (if, for instance, you needed to change the parity setting on that LU). Either way, KERMIT-RTE will enter the "local host" mode, which means that it is the machine that will be the main control for the file-transfer operations. (When you first start the KERMIT session, you are told that KERMIT is in "remote-host" mode, which means that it expects to be controlled by another machine; it is not the master of the communication link.) Once you have set the LU for the remote system, KERMIT will remember it; if you need to connect again you don't need to specify the LU again.

If you are running on an RTE-6 session system, the LU you give must be in your session. Under RTE-A, LU numbers greater than 63 should be accessible, but this has not yet been tested. On either system, KERMIT must be able to "lock" the LU in order to be sure that no other processes can interfere in the file transfer. If KERMIT cannot lock the LU, it will tell you which program has locked it. (Under RTE-6, you can ignore the "/nn" [nn is a number] which appears in the message; it only applies to RTE-A systems.) The LU you give in this command must correspond to a configured 12040B/C or 12792B/C MUX port.

STARTING THE KERMIT SERVER

SERVER

This command causes KERMIT-RTE to act as a server, getting all further commands from another KERMIT in "packets". This command may only be used if this KERMIT-RTE is the "remote" KERMIT and the "local" KERMIT knows how to talk to a server (not all of them do!). Once this KERMIT

becomes a server, you will be told to ESCAPE back to the local KERMIT.

Once acting as a server, KERMIT-RTE may only be shut down by a local KERMIT command such as FINISH, BYE, or LOGOUT, as appropriate. FINISH will shut down KERMIT-RTE but not log-off the session. If you need to rename received files or direct them to a particular directory during a server receive, you may SET RMASK to accomplish this -- see SET RMASK for info. KERMIT-RTE is set up so that if the default communication parameters are already compatible with your PC, you can "[ru] kermit SERVER".

The server mode relieves you of the restrictions of terminal-emulation. The SERVER command can only be used on a "remote" KERMIT, and only if it supports the server mode. The KERMIT which communicates with the server must also be able to send server commands (get, send, finish, bye, etc.). KERMIT-RTE can both act as a server and send commands to a server. The server mode allows you to run the remote KERMIT with "remote control;" once it is serving, the KERMIT looks for it's commands in special packets. A KERMIT server may only be shut down using a finish, bye, or log-off command at the local KERMIT; if the server mode is established on a KERMIT-RTE via a transfer-file, any commands appearing after the server command will not be performed.

When you send files to the KERMIT-RTE server, it will put them into whatever working directory is active, or into file-manager space if no working-directory is active. If this is a problem, you should use the SET RMASK command to specify which directory-path should be used instead. This file mask works exactly like the optional file-mask you might otherwise use in a RECEIVE command in that if a file-name is specified by the mask, that name will be used (in addition to the other information present) for the first file received in each group, and all but the file-name will be used for all succeeding files in the group(s) sent. Because you are not currently allowed to change the receive-mask once the server is active (KERMIT-RTE has no way to get that information in a packet, yet...), sending more than one group of files will be aborted by the server because the named file will already exist, unless you have SET WARNINGS OFF, which will in effect cause the first file of all previous groups to be overlaid! (That doesn't make much sense, does it?) For this reason, a file-name should never be given in a SET RMASK command.

HOW WELL DID THE LAST TRANSFER GO?

STATUS

STATUS causes KERMIT-RTE to display retry- and overall packet-counts and timing information about the most recent file transfer, and retry- and overall packet-counts of all file transfers done during the current KERMIT-RTE session.

The STATUS command tells you about the most recent file-transfer operation performed, giving information like how many bytes and packets were sent in each direction, how much of that was overhead, how many retries there were (they count as overhead), what the average packet sizes were in both directions, how long it took to do it, and the effective baud rate of the transfer. This number is obtained by dividing the number of data bytes transferred (which include the file's name and any quoting characters added, and doesn't include characters removed by data compression) by the amount of time it took to transfer them. Rates of up to 50% of the actual baud rate are not unusual with 94-byte packets, which is the largest you can get in this protocol. As you transfer more files and/or larger files in a group, your throughput will improve.

KERMIT-RTE also keeps track of all packets and bytes sent in both directions since it was started, in case you happen to be interested.

MODIFYING KERMIT'S PARAMETERS

```
SHOW
SET <param> <value>
```

SHOW and SET allow you to see or set system-dependent characteristics.

SHOW causes KERMIT-RTE to display the values of the SET parameters, and various other information.

SET commands require one of the following <parameter-names>:

Binary	BQuote	Check	DEBug	DElay	Escape	Ibm	Line
PACKet	PARity	Prompt	Quote	REpeat	REtry	RMask	Sync
Warning							

NOTE: The minimum allowable abbreviation is shown in UPPER-CASE.

SET commands may also require a value which is dependent on which parameter is being set. If a parameter requires a numeric value, you may enter it in decimal, octal (post-fixed with a 'B'), hexadecimal (post-fixed with an 'H'), or as a single literal character (post-fixed with a '"').

For more information on the settable parameters and the type of allowable values they require, do "help set <parameter-name>" from the above list; to get help on set debug, for example, you would type "help set debug".

It is possible that you could work for years with KERMIT programs for all machines and never need to change a single parameter. If you are a "C" programmer, however, you will probably want to at least change the character used by KERMIT for control-quoting ("#") in order to reduce overhead. If you need to transfer a file which contains a large number of tildes or ampersands, you will probably want to change the repeat-count or binary-quote characters, respectively. If the line is a little noisy, you might want to reduce the packet size and thereby reduce the likelihood of a retry. All these things can be done with the SET commands available in most KERMIT programs. The SHOW command displays the current values in the user-adjustable parameters, in case you forget whether you have already performed a given SET command.

All SET commands require the name of the parameter to be changed. These may be abbreviated to the first 1-3 characters of the name as shown above. In addition, most SET commands require a value to assign to the parameter. If a parameter requires a numeric parameter, it may be entered in decimal, octal, hexadecimal, or as a literal character; you may use whatever mode is most convenient. As an example, if you needed to change terminal-emulation escape character to a control-\ (abbreviated as ^\ in most KERMITs), then

SET E 28	(decimal, because there is no trailing character)
SET E 34B	(octal, because of the trailing "B")
SET E 1CH	(hexadecimal, because of the trailing "H")
SET E ^"	(literal, because of the trailing quote)

are all equivalent ways to set the escape character to an ASCII "FS." The benefit here is that you can probably remember (in the context of how you use the escape character) the

control-sequence you need to use better than the character-code needed. It wouldn't make much sense to try to remember that the character code of a caret is 94 if you were wanting to change the repeat count character to a caret, so KERMIT-RTE allows you to enter it as a literal. Similarly, it wouldn't make much sense to remember that 94 is the character code for a caret if you were going to set the packet size to 94 bytes.

SET BINARY ON or OFF

As of version 1.97, KERMIT-RTE can transfer non-ASCII files to compatible KERMITs if you SET BINARY ON.

"Normal" (non-binary) transfers convert the logical END-OF-RECORD in a file to a <CR> <LF> sequence which all KERMITs know how to use. If this sequence appears as part of the normal data in a file, the destination KERMIT will start a new record at that point and drop both characters. Once you have SET BINARY ON, KERMIT-RTE will transfer all data as it appears in the file and ignore the "special" significance of CR/LF within the file. Further, the file's record-structure is ignored; the file operates as if it's records are all 256 bytes long. For this reason you should GET or RECEIVE (or SET RMASK) using a FULL FILE DESCRIPTION, including the type, size, and record-length. Otherwise the file will default to type-4, 24 blocks, record-length of 0. NOTE: if the communications line requires parity other than NONE, and if you have disabled binary-quoting (see SET BQUOTE), transfers of binary files will not be allowed.

We have been highly successful transferring KERMIT's relocatables to a VAX, and then to another HP-1000 machine using binary transfers. The problem with KERMIT's normal mode of transfer is that if a carriage-return character is immediately followed by a line-feed character, the KERMIT protocol says that a logical record has been ended. This condition may not happen very much, but if it happens at all then the file will not be transferred intact. By setting binary mode on, you do two things:

- 1) Disable the recognition of the end of a logical record, and
- 2) Force KERMIT-RTE to treat all files as if they were type-1 extendable, except for files which are received explicitly as type-6, which may not be extended.

KERMIT-RTE doesn't need to worry about logical records when it is sending or receiving a type-1 file because it is transferring the "disc image" of that file. This has certain side-effects of which you must be aware. Any unused space at the end of a file will be transferred, because KERMIT-RTE will not recognize the logical end-of-file; this adds some overhead to the transfer. The file(s) received must be renamed with the full specification of the type, size, and record-length if they are not type-4 with 24 blocks for the first extent. If KERMIT creates the file (rather than overlaying it) then if you always create the file arbitrarily larger than it is, KERMIT-RTE will truncate the part into which no transferred data was written, creating a file with no extents. Since the file is created without regard for it's logical structure, the CI information regarding the size of the largest record, how many records, where the EOF is, etc., will not be available, but this is not usually a problem.

If you SET BQUOTE 32, you will turn off binary-quoting. This is not a problem unless the communications line over which the file is to be transferred is using the parity bit. If the line parity is something other than NONE, and if binary-quoting is disabled, KERMIT-RTE won't let you transfer in binary mode because it would not be able to transfer the most significant bit of each

byte.

Before you use binary mode in KERMIT-RTE, be sure that the other KERMIT will support it. In the VAX KERMIT, you must SET FILE TYPE FIXED to use the binary mode in KERMIT-RTE; the VAX KERMIT also has a SET FILE TYPE BINARY which is not compatible with KERMIT-RTE's binary mode. Even if you use SET FILE TYPE FIXED on a VAX KERMIT with the binary mode in KERMIT-RTE, you will get an error-message anyway if a file with an odd number of blocks is sent from the RTE system to the VAX, because it's minimum block is 512 bytes to RTE's 256.

SET BQUOTE <value>

If you need to send non-text data to another KERMIT using a communications line with parity other than NONE, KERMIT-RTE can do "8th-bit prefixing" if the other KERMIT agrees to do it also. What this means is that if KERMIT is about to send a character which has it's 8th bit set, KERMIT will send a special character (the "BQUOTE") before it (which tells the other KERMIT to set the 8th bit on the next character it receives). The setting of the 8th bit would otherwise be lost due to the parity setting.

With this command you may set the BQUOTE to something other than the usual "&" (46 octal). The value you enter is the character-code of the character you would like to use (which must match what the other KERMIT expects); it must be in the range of 33-62 or 96-126 (all numbers decimal) and it must be different than the QUOTE (for control codes) and the REPEAT (use for repeat- count processing). You may enter this number in decimal, octal (nnnB), hexadecimal (nnH), or as the literal character followed by a quote (").

You should not have to change this for most KERMITs. Setting the BQUOTE to 32 decimal (ASCII blank) turns 8th-bit prefixing off; this interacts with the communications-line parity and may prevent binary-file transfers.

If you never try to transfer binary data (defined for our purposes as information which may have the 8th bit in the byte set), you should never have to worry about the BQuote character. If you transfer binary data, but the line parity is always NONE, you may want to disable binary-quoting completely. Otherwise, BOTH of the KERMIT programs must agree to do binary-quoting, and they must use the same character when they do it. That is where this command comes in. If you know that the "other" KERMIT's BQuote is different from KERMIT-RTE, you can change the BQuote used by KERMIT-RTE. Be sure to read the information about the SET BINARY command and the implications of disabling the BQuote.

SET CHECK <checksum type>

This command allows you to request alternate checksum bytes within packets:

- "SET CHECK 1" builds 1-byte (arithmetic) checksums <default>
- "SET CHECK 2" builds 2-byte (arithmetic) checksums
- "SET CHECK 3" builds 3-byte (CCITT-CRC) checksums

By setting CHECK to 2 or 3, KERMIT's error-detecting capability increases, but only if the other KERMIT can also do it! If it can't do

2- or 3-byte checksums, don't worry, because all KERMITs already know to do the 1-byte checksum.

KERMIT's 1-byte checksum actually is very effective. If you are worried that a noisy communications line might cause some errors that might not be caught in the 1-byte checksum, you can improve KERMIT's ability to detect errors by setting the checksum type to 2; if you are very worried, you might set it to the 3-byte CRC. The 2- and 3-byte checksums increase the overhead by reducing the number of data bytes that will fit in a packet (not by much, obviously, but it does add up).

When you request the 2- or 3-byte checksums, KERMIT-RTE will ask its partner to do the same type of checksum when it sends to a non-server KERMIT; the other KERMIT will ask KERMIT-RTE to do 2- or 3-byte checksums only if you have given it the appropriate command telling it to ask for these alternate checksum types, or if it accepts an "I" packet as a server. If you don't arrange for both sides to do the alternate checking, or if the other KERMIT doesn't have an alternate checksum implemented, both KERMITs will use the 1-byte checksumming method.

SET DEBUG <keyword>

In the event that you experience some problem with KERMIT-RTE's operations, you may arrange for KERMIT-RTE to perform some self-diagnosis. Before you can debug any of KERMIT-RTE's operations, you must first setup its debug logging file via "SET DEBUG FILE <file-name>", which will create the given file if it doesn't already exist. NOTE - if the debug logging file already exists, it will be overlaid! You may then "SET DEBUG <type>" as shown in the following to obtain:

STATES - shows packet numbers/types and internal state as

A Abort transmission	B Break transmission	C Transfer complete
D Data	E Error	F File header
R Receive initiate	S Send initiate	T Timeout
Z End of file		

PACKETS - shows the actual data in incoming and outgoing KERMIT packets. You will need to consult the KERMIT protocol manual in order to get a description of these

ALL - a combination of STATES and PACKETS

OFF - turns off debugging

The built-in debugging features of KERMIT-RTE were more for of my own benefit during the debugging phase. If you are trying to implement a KERMIT for some machine which currently has none, you can use the debug logs to locate protocol errors or other bugs in the new program, using KERMIT-RTE to send or receive messages as desired. In order for this information to be very useful, you need to be familiar with the KERMIT protocol.

SET DELAY <value>

This sets the time in seconds which the KERMIT-RTE will delay before it sends the first packet of a file. It starts out at 15 seconds and you may change to anything from 1 to 30 seconds, to give yourself time to escape back to the local KERMIT to give a RECEIVE command. This command will not be allowed if this KERMIT is the local one.

SET DELAY 25 sets the send-delay to 25 seconds.

When using the "Send-Receive" method to transfer files from a remote machine (using KERMIT-RTE) to the local one, you must:

- 1) Connect and log-on to the remote machine, then invoke KERMIT,
- 2) Give the SEND command for the file(s) you want,
- 3) Escape back to the local machine, then finally
- 4) Give the receive command.

KERMIT-RTE will wait fifteen seconds after the completion of step 2 above before it will send the first packet of the transfer. Assuming that you have no trouble performing step 3 and you don't need to rename the received file(s), this should be ample time (maybe even too much). If you find that it is taking you much longer or shorter to do steps 3 and 4 above, you may want to lengthen (up to 30 seconds) or shorten (down to 1 second) the delay time.

```
SET ESCAPE <value>
```

The escape character is the character that you enter to "escape" back to the local KERMIT after CONNECTing to another machine. The normal escape for KERMIT-RTE is control-] (35 octal) and probably doesn't need to be changed. If this character is commonly used by the other machine for some reason, you may change by "SET ESCAPE <value>", where <value> is the character-code (a number) of the escape character to be used, which must range from 1 to 31 (decimal). The actual value may be entered in decimal, octal (nnnB), hexadecimal (nnH), or as a literal followed by a quote character (").

The escape character is always a control-code of some kind. It tells the KERMIT terminal emulator to look for the next character as a KERMIT command (rather than sending it to the remote computer). If the default escape character (control-], abbreviated as ^]) is in effect, then

```
^] ^] will actually send one ^] to the remote computer
^] C will "Close" the CONNECTION (return to local)
^] Q will Quit debug logging, if active
^] R will Resume debug logging, if previously active
```

Note that there are no spaces between the escape and the command character.

When any KERMIT is emulating a terminal, it "listens" for keystrokes from your terminal (to send to the remote system) and for data from the remote system (to send to your terminal's screen). An escape character is needed if only to allow you to turn off the terminal-emulation. The "escape character" merely tells the local KERMIT to listen for a local command given by the next character typed (that is why there are no spaces between the escape and the next character). The escape followed by the "c" (in upper- or lower-case) is just one of the commands accepted by most KERMIT's to turn off the terminal-emulator.

An example usage of this command: I occasionally use a Macintosh™ computer as a terminal on our system. It is not unusual for me to want to run KERMIT-RTE to the VAX, but the terminal-emulation program I use will not pass the "^]" (control-]) key, for reasons unknown. To fix the problem, I usually SET E ^B", which sets the escape to control-B.

```
SET IBM ON or OFF
```

You may "SET IBM ON" or "SET IBM OFF" only if the local KERMIT is

talking to a remote KERMIT requiring IBM mode (half-duplex). If IBM mode is ON, KERMIT will wait until it has received a DC1 (XON, or ^Q) character before it transmits anything. Further, IBM ON causes KERMIT-RTE to locally echo keystrokes during CONNECT mode.

I am sorry to report that this is not tested at all. I am unable to find anyone locally who is interested in installing a KERMIT on an IBM computer, or any other machine with similar (half-duplex) communication requirements.

```
SET LINE <lu>
```

This tells KERMIT-RTE which logical unit number (<lu>) is to be used for communications to a remote computer. The LU# must be legal for your session, and should be a 12792B/C or 12040B/C multiplexer port hooked to the desired remote computer or modem.

KERMIT-RTE starts up in "remote-host" mode, which means it expects that you are running from a PC and will transfer files on "LU 1". By setting the line to some other LU, you are putting KERMIT-RTE into "local-host" mode. Some commands work in one mode and not the other, or they operate differently depending on the KERMIT's mode. You can switch from local-host mode back to remote-host mode by SETting LINE to your terminal LU.

This command works like the CONNECT command with the optional LU# parameter included (in fact CONNECT calls the SET LINE code to process the LU# parameter), except that it doesn't actually start the terminal-emulator with the LU. You would use SET LINE rather than CONNECT if you needed to alter the parity of a MUX port before you did connect to it. Note that you will not be allowed to SET LINE to anything but a MUX port, or back to your own LU if you SET LINE 1. When you have completed a KERMIT session after having SET LINE to some other LU, you can avoid the 5-second delay in the termination of KERMIT-RTE by setting the line back to your own terminal.

```
SET PACKET <size>
```

This is used to change the packet size from it's default 94 bytes to any size from 31 to 94 bytes. If the connection is very good, you will get the best throughput with a packet size of 94 bytes. For noisy lines, you can avoid some (costly) retries by reducing the packet size.

With this command you make the most major impact on the efficiency of the KERMIT protocol. The larger the packet size (up to the 94-byte limit, which is the default value for KERMIT-RTE), the lower the overhead, ignoring quoting sequences. If the line is noisy, any packet which has to be retried is entirely overhead, so reducing the packet size under these circumstances would increase the efficiency of the transfer.

```
SET PARITY NONE/ODD/EVEN/MARK/SPACE
```

As of version 1.97, you may set the parity of KERMIT's remote line. When the system boots up, a port's parity is set by the system manager to suit the needs of most users. If you need to communicate with a remote system which uses some other parity, you may change it (before you CONNECT) to be compatible with the other system. KERMIT will

restore the original parity when you SET LINE to some other LU or exit from KERMIT.

WARNING -- use this command with extreme caution! If the parity you set is different from what the other system actually does use, you may still be able to talk through CONNECT, but file-transfers will be impossible, and it will be difficult to determine why!

NOTE -- KERMIT-RTE supports "binary quoting" for the transfer of some non-printable data over lines where parity is not NONE (do H BQuote for info).

This command is not extensively tested - **BEWARE!** KERMIT-RTE correctly identifies ODD, EVEN, and NONE parity if that is the way a port is already set, but it is possible that MARK and SPACE parity have been reversed. If KERMIT-RTE correctly identifies these parity settings, then it will probably do the correct operation in changing them.

SET PROMPT [<up to 20 non-blank characters>]

When you are trying to work with a copy of KERMIT-RTE at both ends of a link, you may have some difficulty in determining whether a command-line prompt is for the "local" KERMIT or for a "remote" KERMIT. To resolve this problem, you may change the command-line prompt that KERMIT-RTE uses to any sequence of up to 20 non-blank characters (a blank terminates the prompt-string); KERMIT-RTE will shift the string to all upper-case. If no prompt-string is given, the original command-prompt will be restored.

If you should try to use the same KERMIT program on both ends of a link, like aVAX-to-VAX or RTE-to-RTE connection, you will discover that other than the slowness of the terminal emulation, it is difficult to determine whether a given command is going to be processed by the local KERMIT or by the remote one. This command can alleviate the problem (for the RTE-to-RTE link) by changing the command-line prompt at either end. It is also useful if you don't like the normal prompt used by KERMIT-RTE.

SET QUOTE <value>

The quote character is the character that the local KERMIT sends to prefix control-characters which may occur in the files to be transferred. The normal quote character is "#" (43 octal), and there should be no reason to change this. If the remote KERMIT requires some other quote character, enter it's NUMERIC VALUE (character-code) as "SET QUOTE <value>", entering the value in decimal, octal (nnnB), hexadecimal (nnH), or as a literal character followed by a quote character ("). The character-code for this alternate quote character must be in the range of 33 to 126 (decimal), and it must not conflict with the BQUOTE character (for 8th-bit flagging) or the REPEAT character (for repeat-count processing).

ALL files contain control characters, if only to mark the end of a record! As noted above, if you want to transfer a "C" program (which uses a "#" more than many other languages), you will probably want to change the quote character, which itself must be quoted if it appears in the data to be transferred.

SET REPEAT <value>

The repeat character is the character that the local KERMIT sends to prefix a repeat-count character. If both KERMITs agree to do it (with the same character), data can be compressed if 4 or more identical characters need to be sent consecutively.

This command allows you to set KERMIT-RTE's repeat character to match the one that the other KERMIT is using. The usual repeat-character is a "~" (tilde, decimal 126, or octal 176), and most KERMITs which know how to do this will use it. If the KERMIT you want to talk with uses some other character to prefix repeat-counts, you may enter it's character code here in decimal, octal (nnnB), hexadecimal (nnH), or as a literal character if entered followed by a quote character ("). It must not be the same as either the QUOTE character (for control-codes) or the BQUOTE character (for 8th-bit processing). The value you enter must be in the range of 33-62 or 96-126.

Like the quote character, you would only want to change this if the file(s) you are trying to transfer already contains a large number of "~" characters, and if the other KERMIT can do repeat-count processing.

SET RETRY <number of retries/packet>

If KERMIT encounters an error in receiving or sending a packet to the other computer, it will retry sending that packet. It will keep on retrying the packet transfer operation until the retry-limit is reached. The retry-counter is reset each time a packet is successfully transferred so that an intermittently noisy line can be tolerated.

KERMIT-RTE starts with a retry-limit of 5; with this command you may change it to any value from 5 to 30. If you can't transfer a packet trying 30 times, you probably never will.

I hope you never need to change the retry-limit. 30 as an upper-limit for this value is purely arbitrary. If you decide you need to retry even more often, you can alter the code in the SetRetry subroutine; the time is yours.

SET RMASK <file-mask>

Since server-receives won't allow you to specify any receive-mask info, this command, given before the server command, will do it for you. The file mask may contain specifications for directory, subdirectory, type-extension, file-type, file-size, security-code, record-length, and so on, but wild-card characters may not be used. NO CHECKING IS PERFORMED as to the validity of the file-mask when it is entered, only when it is used. If the file-mask parameter is not given, a previously-defined file-mask (if any) will be cleared; otherwise, any subsequent server-receives will function exactly like a RECEIVE command with the same file-mask. Do "HELP RECEIVE" for more information.

The receive-mask (RMASK) does for the KERMIT-RTE server what the second parameter in a RECEIVE command does for the local KERMIT-RTE: rename the first file in a group (if a file-name and/or type-extension is given) and give a file-path for all of the files in that group (if the file-path is

given). It is not a good idea to actually specify a file-name in the RMASK, because in addition to this, it performs these actions for all groups received by the server. I intend to add "remote" commands to KERMIT-RTE in the future, which should help to alleviate this kind of problem.

SET SYNC <value>

The sync character is the character the local KERMIT expects to receive as the first character of any packet from the other KERMIT. Control-A is the default sync for most KERMITs and there should be no reason to change it. If the other KERMIT uses some other SOH, you must "SET SYNC <value>", where <value> is the character-code (a number) of the sync character to be used; it must be from 1 to 31 and must not conflict with the EOL or IBM-prompt characters.

The character-code may be entered in decimal, octal (nnnB), hexadecimal (nnH), or as a literal followed by a quote character (").

You would only want to change the "sync" or "mark" character if, for some reason, the other KERMIT had some problem with the default control-A and had some other sync implemented.

SET WARNING OFF or ON

You may "SET WARNING OFF" to allow received file(s) to overlay existing file(s) with the same name.

You may "SET WARNING ON" to restore KERMIT-RTE's starting condition, in which a file-reception which would overlay an existing file will be aborted and you will get an error message telling you about it.

We in the HP-1000 world don't have the luxury (or the headache) of multiple versions of a file (in the same directory). If we did, this command wouldn't be necessary. KERMIT-RTE does not have the ability to automatically rename a file (it can do so manually via SET RMASK or by the second parameter to the RECEIVE command), so you choose: overlay the file or abort the transfer.

RUNNING ANOTHER PROGRAM WHILE INSIDE OF KERMIT

"RUN [XQ] <program-name> [<run-string parameters>]" causes KERMIT to schedule the named program with wait, passing any run-string parameters as supplied. Due to the nature of KERMIT's command parsing, it is not possible to pass lower-case characters or commas in the run-string. KERMIT will inform you of any scheduling errors if they occur. If the program is scheduled successfully, KERMIT will report back any return parameters (in decimal) and/or any return string.

Let's say that you are in the middle of receiving a file group, and KERMIT-RTE aborts the transfer because the 5th file already exists. In the older versions of KERMIT-RTE, you had to shut down the other end, disconnect, and wait for KERMIT to terminate before you could examine the local file. Or perhaps there were some retries during the transfer and you want to see if the file is OK. You can now run any local program without leaving KERMIT. When KERMIT executes a program while in local-host mode (meaning: you have SET LINE to some LU besides your own terminal), KERMIT maintains the lock on the LU to the remote system, so that LOGON won't be bothered. If you use the optional "XQ" (after the RUN part of the command, and before the program name) you can even run the program without wait under RTE-A.

USING COMMAND FILES

Transfer <file-name> [NOecho]

This causes all further commands to be obtained from the file-name given. If 'NO' or blanks appear after the file-name, commands obtained from the file will not be echoed at the console; anything else will cause those commands to be shown at your terminal as they are processed.

Transfer-files may contain any legal KERMIT command except the transfer command itself. If the transfer-file contains a blank line, control will return to your terminal for one command, and then return to the transfer-file. If you should inadvertently put a blank line in the transfer-file, you will have to enter SOMETHING at the KERMIT-RTE prompt; entering one or more commas will return you to the transfer-file. At the end of the file control automatically return to your terminal. NOTE: all commands which appear after a SERVER command in a transfer file will not be processed.

KERMIT-RTE supports a single level of transfer-file, for those who need a set routine for transferring files. Each blank line in the transfer-file causes KERMIT to request one command from the user's terminal. If you decide that you don't want to give some command while within the transfer-file (you just want to transfer back) just enter a single comma. You can use a "canned" set of commands to do everything but send characters through the connect mode. If the transfer-file ends without an EXIT, QUIT, or BYE command (which would terminate the local KERMIT), all subsequent commands will once again be obtained from the user's terminal.

REMOTE CONTROL FROM EUROPE OF A TELESCOPE IN SOUTH AMERICA

Gianni Raffi
European Southern Observatory
D-8046 Garching (Munich)
West Germany

INTRODUCTION

The European Southern Observatory (ESO), an institute comprising eight European countries, runs a large astronomical Observatory, consisting of 13 telescopes, at La Silla in the Chilean Andes, 600 kms north of Santiago.

While astronomers today normally travel from Europe to Chile to observe, remote control observing facilities from the ESO headquarters in Munich, West Germany are currently under investigation.

During an experimental remote control (RC) run carried out from Munich, a 2.2 m diameter telescope in Chile, including its instrumentation, was interactively controlled and used by a number of astronomers for their observations in remote mode.

The configuration included two HP 1000 computers connected via a leased telephone line.

THE TELEPHONE LINK

The RC set-up consisted of two HP 1000 computers: one was the control computer of the 2.2 m telescope at La Silla, Chile, (remote computer), and the second was located in Munich, Germany, (local computer = near the user).

They were connected via a leased telephone line, with a standard bandwidth from 300Hz to 3.4 kHz, available full-time to ESO over the test period.

The telephone line consisted of the following trunks, starting from the La Silla side:

- a 500m cable from the 2.2 m telescope to the Chilean Post repeaters at La Silla,

- a microwave link between La Silla and Santiago in Chile,

- a satellite link on Intelsat over the Atlantic up to Germany, leased ground lines to ESO Headquarters.

REMOTE CONTROL CONFIGURATION

=====

LA SILLA 2.2M TELESCOPE (REMOTE SITE)

GARCHING SET-UP (LOCAL SITE)

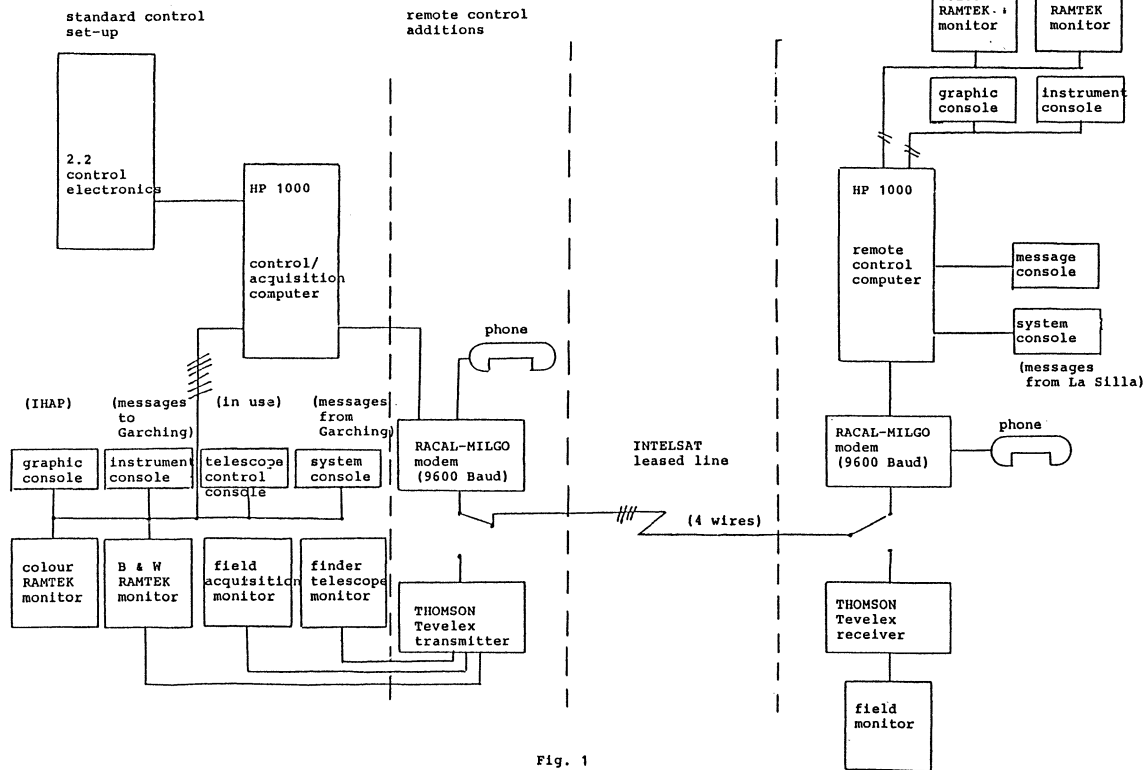


Fig. 1

The line was a 4 wire line, conditioned (equalised) in Munich and Santiago, to provide good quality transmission.

The digital communication was via 2 modems, with adjustable speeds up to 9600 baud, belonging to ESO. It was permitted to have our own modems on the leased international line and this solved the problem of different "quasi-compatible" modems, which is what the German and the Chilean Post would offer.

The communication was point to point, full-duplex and the protocol used was HDLC.

The communication software used at system level was DS/1000 and the ESO software was developed over it.

THE SYSTEM CONFIGURATION

Fig. 1 gives a complete picture of the RC configuration.

It can be seen there how the leased line was shared, so that it was possible to have telephone communication and analog image transmission additionally to digital data transmission, by means of a manual switch.

In particular telephone conversation was possible via special telephone sets connected to the modem.

Analog image transmission was possible via a system to transmit analog video frames, built by Thomson. This was connected to the field acquisition monitor and finder telescopes to send a reduced TV image to Munich in a very short time (25 sec).

It should be noted that the use of the line in an analog way (either for telephone or for video frames) was an alternative to computer communication.

So this could be done only when no data transmission was in progress, but computer operations could later continue unaffected.

THE RC SOFTWARE

1. IN GENERAL

One aim of the RC software was to be able to offer exactly the same interface to the 2.2 m telescope instrumentation from Munich as is available in La Silla (e.g. same softkey menus and forms as on the 2.2 m instrument console).

It was instead preferred to leave direct telescope control via the



Fig. 2

REMOTE CONSOLE SET-UP

=====

From top left:

Field Acquisition TV Monitor, TV Images Receiver, Modem,
Phone, Console to send Messages,
B & W Image Display Monitor, Console to receive Messages,
Colour Images Display, Instrument Control Console,
Graphic Console.

telescope control console to the night assistant in La Silla, following the usual scheme of work at the telescope, where interactive control is shared between night assistant and astronomer. Fig. 2 shows the RC room for the 2.2 m telescope as set up in Munich.

Fig. 3 shows the RC software in a general way.

At the top the ESO control/acquisition software is shown, where the control functions, dealing with the interface electronics and user end part, are implemented in separate packages.

The two parts communicate via mailboxes.

At the bottom of Fig. 3 the same modules communicate under RC via remote - rather than normal - mailboxes. On the user side, while the user end interface was kept unchanged, some interface subroutines had to be added.

2. THE ESO CONTROL / ACQUISITION SOFTWARE

The detailed structure of the various software packages involved in the control/acquisition of the 2.2 m telescope is shown in Fig. 4.

The connecting paths between modules (programs) are implemented with Class I/O and concern the command/reply mechanism.

User end programs make use of function keys and forms (block mode from terminal). This offers the astronomer an interface which is easy to use, but at the same time demands considerable disc I/O.

The information exchanged with the controller programs, directly dealing with the interface electronics, is instead limited to a few simple messages. An example would be the setting of an instrument requiring a dozen motors to be positioned. This has to be explained at length in a form to the user, but when it comes to actually moving the motors, 12 values do the job.

This is why it is advantageous to have the user-end parts and their associated files (for form descriptions, function key menus etc...) as separate modules from the actual control software.

The two parts have a simplified class I/O interface and are suitable to be put on two different computers, provided that a replacement for class I/O exists over DS/1000.

Fig. 4 shows also the cut points between user end and controller - programs.

Fig. 5 shows in detail the structure of the RC version of the same programs, where they have been split into local and remote programs on two different computers.

REMOTE CONTROL SOFTWARE

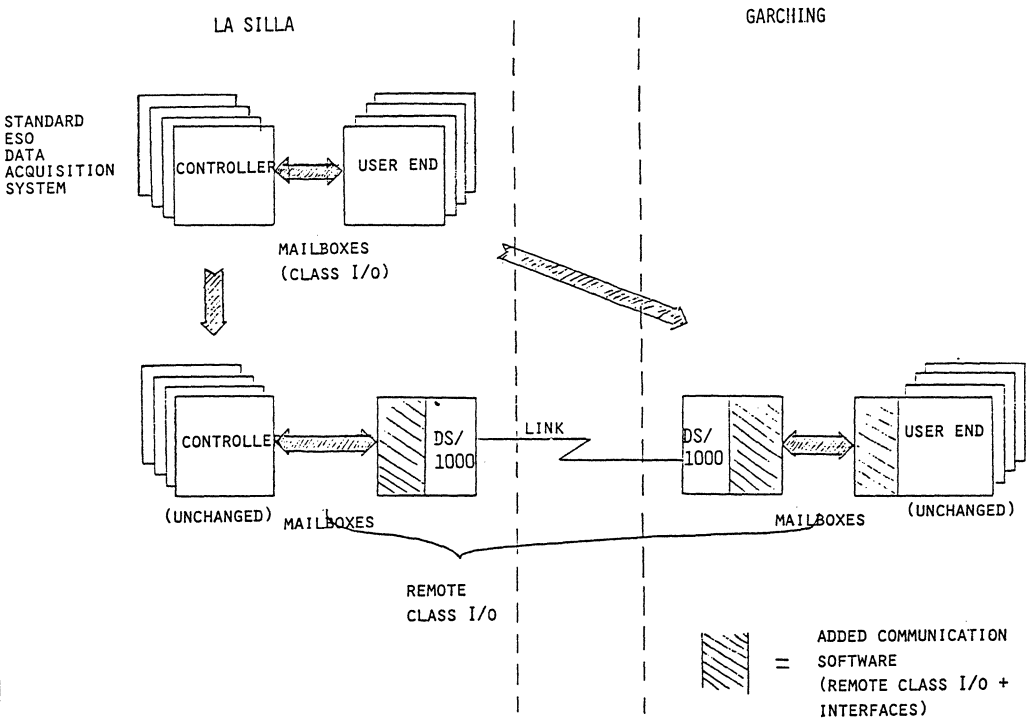


FIG. 3

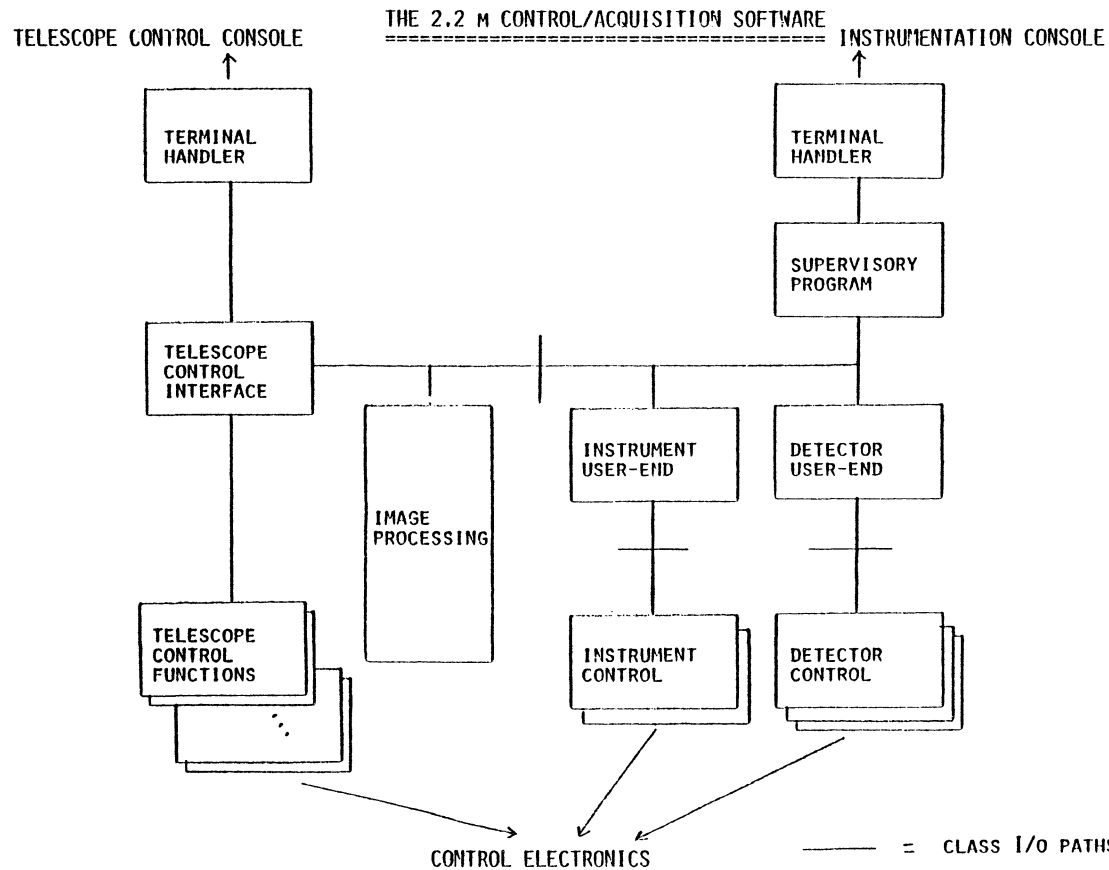


FIG. 4

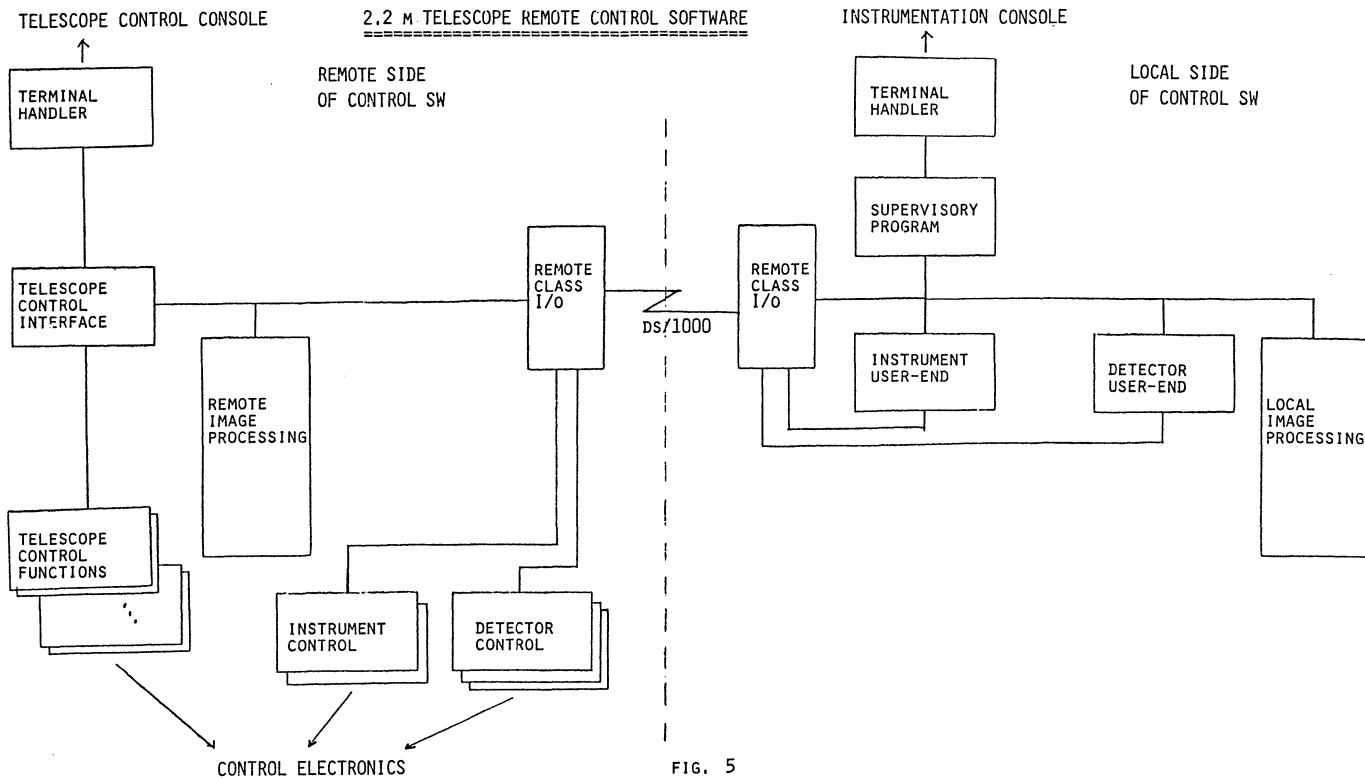


FIG. 5

3. REMOTE CLASS I/O

As DS/1000 did not offer the possibility to do class I/O between two different computers, this had to be added.

The way class I/O is used is so that programs stay waiting on a class until a message arrives. Every program has one class used only for input.

Replies/acknowledgments to a command are returned to the input class of the program, which sent the command.

To keep the control software unchanged under remote control, programs have to do always class I/O to local classes and have to be unaware of the fact that every message may be going to another computer.

Fig. 6 shows how Remote Class I/O was implemented.

DS/1000 offers master-slave program communication, which is, in other words, a one-way channel on which messages travel from master to slave.

The Remote Class I/O package associates classes on one computer to classes on a different computer, so that every message sent to one class is delivered by the package over DS/1000 to the corresponding class on the second computer.

For efficiency (speed), classes are grouped in two clusters, depending on the direction of messages, so that two one way channels implemented with master-slave programs are operated in parallel.

This is shown in Fig. 6, where at the top messages travel from the left to the right hand side and at the bottom in the opposite direction.

In the implementation described here the programs RC Master 1 and RC Master 2 continually poll their clusters of input classes to look for incoming messages to be delivered to the other computer.

Alternatively it would have been possible for master programs to wait on one class only, where then incoming messages should have been tagged with the different destination class numbers.

The implementation described here demands that classes on two different nodes are "connected" with an initialization call, which establishes the connection between the two and the direction of messages. Class numbers can be connected in different ways:

given class numbers on either node can be connected either to a new class number on the other node (to be asked to the operating system there) or to a pre-existing class number.

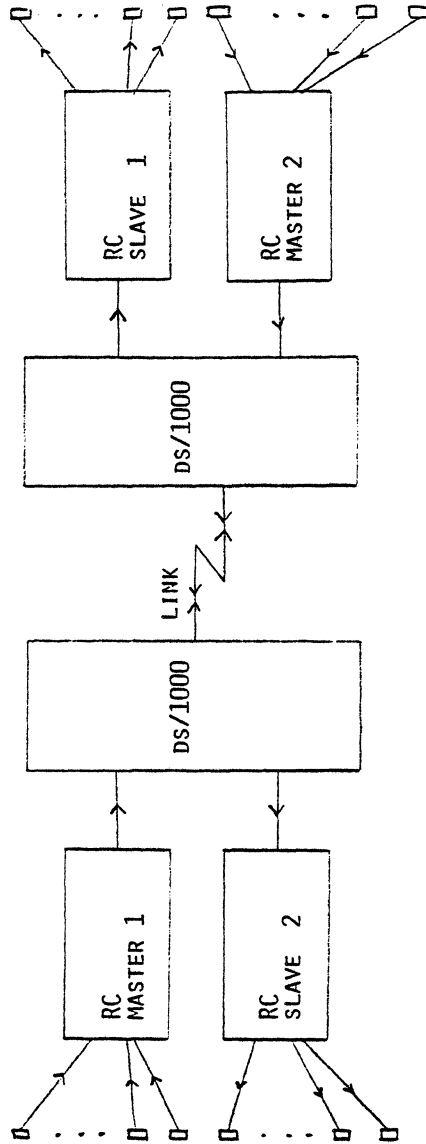


FIG. 6

Considering again Fig. 3, it can be seen that the user end programs have been enlarged with some subroutines for RC. These subroutines do the connections mentioned above at class allocation time. This means, in other words, that at class allocation time the Remote Class I/O programs are started and establish their threads among class numbers on the two computers.

The calling of the connect subroutines in the user end programs is the only addition for RC. In practice this is a portion of code, executed conditionally when a node number for a remote computer connection is given.

In this way the same program can now be executed in local mode and simply no connections are done, whereby the Remote Class I/O package and DS/1000 remain inactive.

4. DATA FLOW

Following the data flow in Fig. 7 it can be seen that data were acquired on the control computer of the 2.2 m telescope (according to commands sent from Munich).

Afterwards data were sent from La Silla via a compression/expansion package. Data went directly from an image processing file in La Silla to a corresponding file in Munich, with an option to display them on an image display monitor.

It should be emphasized that this activity could be carried out in parallel, both with the sending of new commands and with local image processing. This means also that transmission times were much less noticeable than if the user had been idle and waiting.

RESULTS

The leased line, once set-up and equalised, was reliable and stable in time. There were no hang-ups.

The DS set-up was not obvious for such a line. In particular the HDLC cards had to be set at a very low baud rate to achieve longer timeouts, and anyhow the actual baud rate is imposed by the modem.

DS statistics gave negligible error/retransmission rates. Maladjustment of some parameters or bad line quality would substantially increase these rates and transmission of long files would be unreliable.

Disturbances (cross-talk) on the line producing high retransmission rates were measured only once/twice for about one hour in a week.

DATA FLOW =====

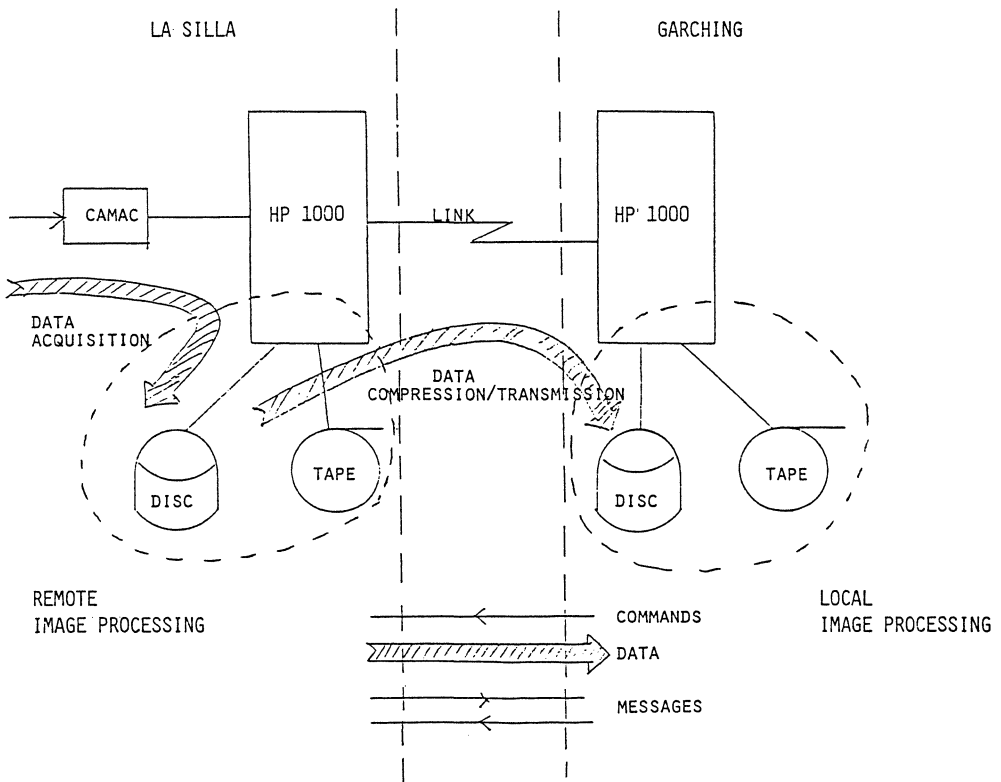


FIG. 7

About global reliability, very few restarts of the control software environment were ever needed. Even short accidental interruptions of the line, by using the telephone when data transmission was going on, were recovered.

The line was operated at 9600 baud.

The transmission of a full CCD frame (around 170KW) took about 10 mins.

Typical data transfer times were however smaller: 7 min for images, 2,5 min for spectroscopic data.

This is because data compression/expansion was used and, for spectra, only the relevant part of an image was sent. Data were sent complete (i.e. without loss of information) in the given times.

Out of 7 min total time, 2 min went for the data compression/expansion algorithm, whereby an image was typically cut down to 50% of its size - so CPU time (on a loaded HP 1000 F CPU) did play quite a role. This would obviously be much better on either an A900 or a Precision Architecture computer.

Altogether the net throughput was some 40-45% slower than in the laboratory tests. This depends on the measurable delays introduced by the satellite link.

The large amount of data transmitted plus the additional advantage of free telephone calls during the day time, make a leased line a preferable solution with respect to packet switching from an economic point of view.

Reliability, reduced delays and all-time availability of a leased line are additional bonuses.

The relatively high cost of the line, about US \$ 500/day, is still reasonable when compared to the costs of astronomers trips from Europe to Chile.

CONCLUSIONS

The success of this test will mean considerable encouragement for ESO, which is investigating the possibility to offer RC as a permanent feature on new telescopes.

The performances of DS over a leased line are quite acceptable for an interactive control system like the one described and in practice no difference with local control is noticeable to the user.

The long transmission times are instead clearly a bottleneck for bulk data transmission, where data compression has to be applied to improve net throughput. The limit to the complexity of data compression algorithms is naturally given by the compression/expansion times, not negligible in comparison with transmission times.

On the whole, DS/1000 and HDLC cards proved to be extremely reliable even in such an extreme case and an improvement on net throughput, due to faster compression/expansion, should be expected with the newest models of the HP 1000 family.

I wish to acknowledge the relevant contribution to this project of my ESO colleagues P. Biereichel, W. Nees and M. Ziebell.

Charles N. Small
Syslog Inc.
4996, Place de la Savane
Montreal, Que. Canada H4P 1Z8

Abstract:

A global automated methodology (SOS) was successfully applied to the salvaging of software from an obsolete hardware environment and its conversion to and implementation on an HP-1000. The application was an automated circuit-board testing system with customized test stations, interfaces and operating system, and specialized languages for writing test programs.

The SOS automated tools support development of hierarchical systems using an integrated methodology applicable from specification through design, coding and testing to maintenance. Using SOS, a system is decomposed into processes, where each process is specified as a strategy executing within an environment. Strategies and environments are structured from primitive components which are stored relationally in a database. The relational system model makes possible rigorous analysis and verification before coding begins and provides detailed metrics during development.

Source code is produced directly from the system model using automated code generation. This provides a high level of language independence, enforces consistency between specification and code, and ensures complete and accurate documentation.

Introduction:

This paper describes the application of an automated methodology to a medium-size software development project. Section 1 outlines problems of software development in general terms; section 2 provides specifics of the particular application project; section 3 outlines the methodology and describes the computer-based tools which implement it. Section 4 outlines some of the benefits which we perceive in using development tools of this type. Section 5 then outlines, for this particular project, each stage in the application of the methodology, from requirements definition to completion; and section 6 provides an informal evaluation of the tools and some observations regarding the process of converting to a new methodology.

1. An Integrated Approach to Design & Development:

Most software development over the past fifteen years has been guided by the so-called "waterfall" model. Defects in this approach have recently been widely recognized [21]. One problem is the use of distinct methodologies for particular stages of development - typically, one for requirements and specification, another for design and coding, perhaps another for testing. Because interfaces between the discrete stages are not well defined, translation must at some point be performed manually, and errors can creep in.

Also, it is difficult to reflect changes made at a late stage back onto the initial specification. This results in inconsistencies.

Another disadvantage to the "waterfall" model is the long time lag often entailed between design and testing. This makes errors difficult to detect (because the system under test is already so large and complex) and expensive to correct. Design flaws are very costly to change at this stage.

In addition, traditional methods of software design have been inadequately automated, in the sense that too few computer-based tools for software design and analysis have been at the disposal of systems analysts. In designing complex systems, it is clearly desirable to relieve analysts and programmers of as much of the burden of repetitive tasks as possible and to provide as much computerized checking and verification as possible at an early stage of design.

In our design environment at Syslog we have been very conscious of these problems, since our work involves complex and specialized automation and control systems in application areas as diverse as water purification systems, steel plant automation and satellite telecommunications. This requires rapid mastery by our staff of specialized needs. Since many of these systems are developed for installation at geographically remote sites, they must be extensively developed before on-site testing can be undertaken. We therefore have been interested in finding and applying development methodologies which are broadly applicable and flexible, which provide a high measure of reliability in the end product, and which allow accurate estimation at any stage of the overall size and detailed status of a development effort.

It was in this context that we began to introduce SOS (Strategy-Oriented System) methodology on certain of our development projects [15]. So far, two medium-size projects have been completed on which SOS was applied from start to finish. One is described in this paper and the second elsewhere [14]. Since the approaches to design required by these two projects were so clearly distinguishable - in fact, almost diametrically opposed - we felt they would constitute a good test of the method's flexibility.

2. The Application System

In the project described here, SOS automated tools were applied to the salvaging of software from an outdated hardware environment and its conversion to and implementation on a newer machine (an HP-1000). It should be emphasized that this is far from being the only possible application of the tools. It is not even true that salvaging and conversion was necessarily the best course to have adopted in this particular situation.

However, many enterprises are today in the position of having heavy investments in existing software systems which run on, and are dependent on, hardware that will be obsolete several years from now. These software systems in many cases embody knowledge that is invaluable, because it is so specific to that enterprise and its particular needs.

Enterprises that depend on some existing systems that provide functionality to meet highly specialized needs will be understandably reluctant to simply scrap these systems and "start from scratch" because the systems are insufficiently portable and their hardware is aging. Knowing the unpredictability of the software development process, they may well fear investing large sums with inadequate results. They may instead choose to salvage the existing system piecemeal, knowing that even though the process may be more difficult, they will be able throughout the process to retain a system that meets their needs.

Salvage and conversion in this sense is thus an important but little-studied problem. In a more general sense, however, there are probably few projects that actually begin with a completely clean slate. In most cases, there are components, procedures, modules, design approaches, which were evolved elsewhere and which it is felt should be retained or adapted with modifications in a new environment. What is to be shared may be, at one extreme, a complete system design, or, at the other, simply file structures, I/O routines, or user input screen formats.

In any case, traditional top-down design approaches deal inadequately, in most cases, with the integration of these pre-existing components.

For the application discussed here, salvage involved implementing a system so as to be functionally equivalent to an existing system. The application was an automated circuit-board testing system with customized test stations, interfaces and operating system, and specialized languages for writing test programs. Software for compilers and test execution facilities was converted from PDP-9 assembler code to HP-1000 C.

The only specification for the system to be implemented (apart from manuals designed for the end user) was the assembler code itself, with its annotations. Installation required that the new system interface to all existing special-purpose hardware and communicate with it over the original PDP-9 bus using the protocols in place, so that all existing test programs might be executed without modification. The assembler code for the PDP-9 system amounted to roughly 10,000 lines.

An additional complication was that none of the original system designers were available. In fact, no individuals at the present installation had any detailed knowledge of the system's internals.

The problem then required working backwards from the executing assembler code in order to extract a specification from it, and then working forward from the specification to its implementation in a new environment. SOS tools were used throughout this process, and using them it was possible at each stage to identify particular components of the existing system with corresponding components of the new system.

In fact, not all of the system needed to be salvaged, since it had been coupled with a time-sharing operating system built in-house, many of the facilities of which were either obsolete (e. g. paper tape read) or provided directly by RTE-A on the HP-1000 (e. g. file system access, user scheduling).

Specifically, the system to be converted included three custom-built hardware test stations for performing specified sequences of tests on circuit boards at the operator's request. One station was operated by hard-coded commands and the other two were accessed through a 16 x 64 x 2 matrix of code points. Test facilities included A/ D measurement (voltage, resistance, current drain, diode forward and diode leakage), and measures of timing and propagation delay, individually programmable by logic family and driver threshold windows.

Test sequences are written in customized programming languages, each program specifying a sequence of tests to be performed on a unit. The language syntax and commands vary depending on the station used for testing. Facilities are provided which allow programs to be edited, saved, compiled and executed. Source programs are compiled to an assembler-like format using opcodes and operands. The compiled programs are interpreted by a test execution facility. During execution, options are available to the operator to specify break points, looping, data display modes, etc.

The hardware test stations were to be retained in the new system, together with the interface through which they communicated to the PDP-9. An additional "black box" interface was built to provide 16-to-18-bit mapping and voltage level conversion. This new interface connected, at one side, directly to the existing hardware interface ; at the other, to a parallel interface card in the HP-1000. An interface driver for the PIC and device drivers for the individual stations were designed and configured into the HP-1000 operating system to handle communications.

The system specification required that the new system provide an execution environment in which all test sequences performed as they did on the current system, and provide, in addition, functionally identical compilation facilities and operator interfaces, and appropriate system management features.

3. The SOS Automated Methodology

SOS (Strategy-Oriented System) developed out of research conducted at Concordia University in Montreal, over the last fifteen years, by W. M. Jaworski and his co-workers [6, 7, 11, 16, 19, 23] based on work in such areas as decision structures [18, 22], relational representations [4] and problem-solving strategies [25]. Numerous experimental applications had been developed during this time, but it was not until Syslog adopted the approach that it was applied to applications of real-world size and complexity.

There are three fundamental principles on which SOS is based:

- 1) Any software system can be represented as a network (usually hierarchical) of PROCESSES, where each process can itself be represented as a STRATEGY executing in some ENVIRONMENT. The strategy can be thought of (roughly) as the control flow of the process and the environment as its data flow.
- 2) Strategies and environments and all the components of each can be represented relationally and stored in a database. This database is then not simply a repository of software, but a tool which allows multiple views of the system to be extracted and verification and consistency checking to be performed at the level of specifications. It also allows code to be generated automatically from the specifications and the constructed environment of the process, once verification is complete.
- 3) The technology is designed to be easy to learn, convenient to use, and "transparent" (meaning that internal functions are easily accessible). This last feature makes the system flexible and easy to adapt for special purposes.

A STRATEGY is composed of CLUSTERS and ALTERNATIVES. CLUSTERS correspond to points during the execution of a process at which CONDITIONS are evaluated and one out of a number of ALTERNATIVES is chosen for execution. The alternative corresponds to some specified sequence of ACTIONS, along with a NEXT cluster and an EXCEPTION cluster. A simple example is shown in Fig. 1.

Ordinarily, after the actions of an alternative are executed, control flow passes immediately to the NEXT cluster, which is the decision point to be evaluated next. By convention, execution of a process begins at decision point 1 and continues until a NEXT cluster of 0 is found. This 0 indicates the exit point.

In the example, A1 will be unconditionally executed on entry to the process. Then, depending on the current state of Condition 1 (true or false), either Alternative 2 or Alternative 3 will be executed. In the second case, execution of the process ends. In the first, it passes to another decision point (Cluster 3). In the case of loops, alternatives will return to the decision point at which they start, as in the case of A4 and A5 in the example.

Note that any action executed as one of the sequence of actions for an alternative may ITSELF be a complex action, or process, executing its own specified strategy in its own environment. In that case, control returns to the higher level process on exit from the lower level one.

Alternatives may optionally be created with postconditions as well as preconditions, to ensure that specified conditions are met before transferring to the next decision point. These postconditions are known as GOALS. When an alternative fails a postcondition test, control passes to the EXCEPTION cluster instead of the NEXT cluster. An entry in the EXC column indicates that such a postcondition check has been specified.

The ENVIRONMENT in which a process executes is specified by OBJECTS (data elements) and OPERATORS which modify objects or test their status. In combination, these two are used to build ACTIONS and CONDITIONS for the environment. Data flow may be determined at the level of an action, alternative, cluster or process by examining the input and output relations of objects in each action and condition.

All components of strategies and environments are represented relationally in a database, which is then used for querying, verification and validation, report generation, and source code generation (Fig. 2 provides an overall schematic view of the development process). Reports may be produced in a variety of formats, intended for designers, project leaders, administrators or end users, showing control flow and data flow for a process, part of a process, or the system as a whole, process hierarchies, project status reports, and so on. These are produced in SOS notation, in structured English, tabular format or graphically.

4. Advantages of an Automated Methodology

We investigated several available development methodologies [1, 2, 3, 5, 8, 10, 12, 13, 17, 24, 26, 27, 28, 29] and found, in general, that most existing tools are designed only for a particular stage of the software life cycle and that they do not fully exploit the potential of automation in providing computer-based tools which may be used throughout the life cycle. Nor do they provide much support in moving from the design phase to the coding phase, or from code back to design, when revisions are necessary.

Newer tools intended to provide such support and validation are beginning to appear [9, 20]. However, they are still expensive, not as easy to use as one might like, and require expensive hardware to run. We see these tools as likely to become increasingly important in the near future. SOS provides, for us, valuable capabilities without exorbitant cost, and, as a further benefit, allows our software designers to become familiar with the tools not only as users but as developers and improvers of them.

SOS, we find, provides a notation in which system designs may be specified simply yet precisely, allowing inconsistencies and vague terminology to be pinpointed and clarified. Using SOS, an abstract model of the system may be designed quickly, partially implemented, and tested at an early stage, when many of its modules are simply "stubs". At this early stage, the eventual end user can interact with the system, approve or disapprove, or make suggestions or recommendations. This process may be repeated throughout development so as to avoid unpleasant surprises in the final product.

There are no gaps between the specification and design phases and the coding phase of development. Once a process has been specified and verified in its abstract form, and an environment has been designed for it, implementation is then simply a matter of coding primitive elements of the environment in some selected source language, be it FORTRAN, Pascal, C or Assembler. Primitive elements are often simply single lines of source code. Once this has been done, code may be generated automatically.

Systems developed may be designed to run on any particular target machine and be compiled using any desired programming language and compiler. Provided there is a means of transferring source code from the development machine to the target machine, there will, in most cases, be only minor changes required in the code generator. In the project considered here, development was done using SOS facilities installed on PC XT and AT compatibles, for implementation in C on an HP-1000. Another project developed FORTRAN code for a PDP-11; in a third (now in progress), assembler code is generated. Identical SOS technology is used in all cases.

We have found it worthwhile to provide each software engineer with an individual PC on his/ her desk, with an individual SOS database and copies of all the tools. The cost of the tools and the individual microcomputers is low enough to allow us to do this. Designers may work independently of the hardware of the eventual target machine, thus minimizing bottlenecks due to hardware and software testing or system overload. It is possible at any stage of development to integrate several individual databases on a single machine to acquire a full project overview.

Using a relational model for software allows the designer to be equipped with tools which permit much analysis to be automated - control flow and data flow may be verified for consistency and completeness, strategies and environments or their components may be copied and reused, graphics displays and diagrams may be generated, and so on. Some of these tools already exist in our SOS systems, and we are in the process of adding others as the need for them makes itself felt.

Software components are entered, viewed and manipulated using screen-based editing facilities with a query language (SQL or the equivalent) interface. Using SOS, it is possible to work at any point in a system, on any process within it, at any time. This allows great flexibility in allocating development resources and determining which modules are to be developed when, and to what level of completion.

In some cases, it is desirable and practical to design and implement a system from the top down - first a shell, command processor, or user interface routine, and then downwards, ending with utility functions, low-level I/O routines, and so on. In other cases, a bottom-up strategy may be preferable, at least for certain components of the system. SOS allows both approaches, or a mixture of them.

It is also often the case that a system is not designed "from scratch", but with the advantage of algorithms, data structures, interfaces, code modules, assembly routines and documentation developed elsewhere, or independently, or for earlier versions of the system. With SOS, components and subsystems can be copied and reused, or simply specified as "external" and introduced at the appropriate point without further decomposition.

At all times during development, it is possible for project leaders to find out quickly how much has been done and where, and to estimate accurately how much remains to be done. The metrics used are number of processes, strategies, environments, actions, conditions and objects, number of environments coded, and number of processes generated and tested. Fig. 3 provides an example of one such status report, generated at a series of dates for one of the subsystems (a compiler) of the project described here.

During testing, SOS code generation tools provide dynamic trace and debugging facilities. The trace may be activated during testing and removed, in the case of each specific process, when its validation is completed. It may be replaced later if problems develop or modifications to the process are to be performed.

We have found SOS documentation tools to be useful both at intermediate stages of development and for completed and installed systems. With SOS, it is possible to ensure that documentation is complete and consistent, since it is produced during development (instead of after it, as is usually the case), and is automatically updated when components are added to the running system, or when existing components are changed or deleted. It is an invaluable feature of SOS, in fact, that system documentation, system specification, and executing code are always in precise correspondence one with another.

5. The Development Process

The project described here involved four major software components: three compilers and a test execution facility. Each used, in addition to the SOS-specified processes, specially designed drivers and a small number of library processes to provide access to HP-hardware dependent facilities. Overall, the system included 270 processes, as follows:

Processes:

Compiler 1	25
Compiler 2	50
Compiler 3	65
Test Executive	80
Libraries	50
Total	270

Approximate totals for components in the delivered system are as follows:

Clusters	845
Alternatives	2375
Objects	2595
Actions	2980
Conditions	1465

The phases through which conversion proceeded were the following:

1) Extraction of assembler into informal pseudo-code. (This was done in the initial stages prior to the introduction of SOS tools; in later stages it was not found to be necessary and SOS strategies and environments were designed directly.) Fig. 4 is a sample of the original assembler code.

2) Structuring of strategies and environments using SOS tools. Strategies were extracted as single-entry and single-exit units of control flow. Environments were isolated using models of internal system data flow. Fig. 5 is a data flow model for the assembler routine of Fig. 4, showing objects identified as global or local, input or output. Fig. 6 shows actions and conditions for this same environment, structured from the primitive objects of Fig. 5.

3) Modeling of complete system as hierarchy of processes, each of which comprised a strategy and an environment. Components of the system are at this stage represented using narrative description. Much of the verification for consistency and completeness of the system takes place here. Global data flow for the system, is assembled from the environments of individual processes, and used for verification. Fig. 7 shows the strategy (control flow) of the assembler routine of Fig. 4.

4) Coding of primitive components of the environments of each process (actions, objects, and conditions) in an implementation language (in this case, C). Code for components is shown in Figure 8.

5) Automated generation of source code for each individual process using the SOS model. Code is generated with embedded dynamic trace capabilities for purposes of testing and debugging. Source code is transferred to the target machine using a communications program, and is compiled and linked on the target machine. Each process in the system is modeled, generated, compiled and tested separately. Verification, in this case, proceeded in a top-down manner through the system. Fig. 9 shows a sample of generated C code for the assembler routine of Fig. 4.

During development, the SOS project database was partitioned between two PCs. At times, additional PCs were used for report generation, code generation, or file transfer. Because design and development was not done on the target machine, it was possible to design processes yet to be implemented in parallel with testing of already coded processes. Using the PCs for development also allowed designers to use friendly and easily-modified screen-based interfaces to the database.

Typically, new processes were added to the running system one at a time, or in small functionally related groups, all with embedded trace and debug code. Corrections for errors or bugs found during testing were entered into the SOS databases on the PCs and processes were regenerated and transferred to the target machine. Provided turnaround time in this process is adequate, it should never be necessary to edit source code manually on the target machine. If this rule is observed, specification, documentation and executing code will always correspond precisely.

From start to completion, the project required 16 months, although the size of the team varied during this period (from 2 members initially to 5 at its peak, toward completion). The table below shows estimated productivity levels in terms of system components at various phases of the development process:

Months:	2	4	6	8	10	12	14	16
Totals:								
Strategies(narrative):	10	40	60	70	80	130	200	270
Environments (narrative):	2	2	2	10	60	120	190	270
Processes (narrative):	5	30	50	60	70	110	180	270
Environments (coded):	0	0	0	10	40	100	160	270
Processes (tested):	0	0	0	0	5	50	150	270

The noticeable speed-up towards the end of the project may be attributed to four factors:

- 1) The largest and most difficult system (the test executive) was built and tested first.
- 2) Not all of the SOS tools were in place at the start of the project, and when they were available there was a learning curve before the team thoroughly understood them and found the best ways of applying them to their particular problems.
- 3) There was, in addition to the design and specification work, much clarification of the hardware environment of the application, implementation language features, and working methodology (file transfer, compilation and linking, version control, test data design and testing methodology, etc.) that needed to be carried out in the early phases of the project.

6. Evaluation and Commentary

Software salvage and conversion is a demanding task, and one which is somewhat unrewarding for the designer: instead of the satisfaction of seeing a new system perform to specifications, the best that can be hoped for is to see an old system still doing what it always did. Moreover, the task is particularly difficult where inadequate documentation exists for the system to be salvaged. Without automated tools (and the challenge of learning how to use and improve them), the task would perhaps have been irretrievably unattractive.

With SOS, it was possible to do a good part of the design work without reference to the original assembler code. Once an initial specification was extracted, returning to the assembler listings was only necessary where difficulties arose.

Mechanical conversion of such a system without grasping the purpose and functionality of each separate module was clearly impossible. SOS was most useful in providing means for arriving at the semantic understanding needed for converting a module. It made it possible to determine what a piece of code was intended to do, quite apart from implementation details. At this level, we used SOS as a knowledge modeling tool.

As an additional benefit, the knowledge extracted in this way is later helpful to the users of the system, as a means of understanding in non-technical fashion what the system actually does, as opposed to what it was designed to do, or what its users generally believe it to do.

Once a reasonable semantic understanding was obtained, two approaches typically presented themselves: to reimplement the existing algorithms so as to approximate as closely as possible the original, or to simplify the algorithms and reimplement only the functionality.

The first approach was tempting where the original code was difficult to comprehend; but problems might arise if the incomprehensibility resulted from exploiting special hardware features of the original machine, or from code that was poorly designed or frequently revised in the original.

The second approach seemed in many cases to promise increased clarity and simplicity, but modifying some piece of code drastically involved the assumption that we thoroughly understood the original. This was rarely the case. In addition, if code in the new system differed too much from the original, it became difficult to map one system into the other during debugging, as we found we frequently had to do.

In most cases, we adopted the first approach, mechanical reimplementation, as being the safer. This worked, on the whole, surprisingly well, in conjunction with the SOS tools. Even complex and difficult modules would commonly perform on the first run with only minor and obvious bugs, if extraction and conversion had been done with care.

Determining what precisely was to be considered a bug presented special problems. In some cases, the bugs we found existed in the original version. In these cases, we did not attempt to correct the problems, but merely reimplemented the bug, leaving improvements for later. In other cases the bugs had clearly been introduced during conversion and the required corrections were obvious. Many cases, however, were on the borderline. For these, it was either difficult to determine what the original system did, or difficult to understand whether what it apparently did was correct. In general, it proved safer to implement the system as closely as possible to the original. There were functions in the original which actually depended for their operation on the existence of some bug in another function.

All source code was automatically generated from the SOS database, except for a handful of library modules for I/O and similar features. The advantages of using generated code are considerable: the elimination of laborious hand coding, drastic reduction of syntax errors and typographical errors, promotion of the reuse of components, and production of a language-independent design.

We found no problem with the efficiency of the generated and compiled code. However, the benefits to be derived from reducing sources of error and ensuring consistency of code with specification depend greatly on the turnaround time for source code generation. Without adequate turnaround time, it becomes tempting to patch source code manually rather than wait for the output of the generator. Initially, turnaround time was fairly slow: roughly fifteen minutes for a process of several hundred lines, on a standard PC XT-compatible. With enhanced hardware, however, we have reduced the time for generation of an equivalent process to one to two minutes, much closer, we believe, to an acceptable range.

SOS provides a high degree of control over the development process. This is particularly needed in large and complex systems. It is possible from data flow reports to isolate groups of modules which may be worked on without introducing side effects elsewhere, or with side effects whose results are known and controlled. It is also possible, using data models, to effect global changes fairly easily, pinpointing all the modules to be affected by a change in a data structure or access method, for example. In this way, it is possible to obtain increased parallelism within a group of designers working on a project.

An interesting side effect of the approach used was the extraction of a hardware-independent design. This occurred because a complete and detailed design was produced before any attempt was made to map actions and conditions onto specific features of the target machine (the HP-1000). As a result, in the final system, hardware dependent routines (those which use RTE EXEC calls or EMA, for example) were coded separately and isolated in specific library modules.

All the programming and design staff at Syslog has by now been trained in the use of SOS. We believe it can provide a useful tool for communication across as well as within projects. For example, knowing the method, it will, we believe, take less time to bring a new project team member "up to speed" than it otherwise would.

It is of course necessary that there be precise and generally agreed-upon definitions of key terms, an inventory of available tools (editors, generators, queries) and their use, and a means of training new staff and, at times, clients, in the methodology. It is too early to determine how many of the tools developed so far will be reusable on other projects, but we believe many of them will be, since their application is fairly general. It is probably helpful to have a staff that is (as is the case at Syslog) generally young and flexible in outlook, without commitments to traditional ways of doing things, and willing to experiment.

We have adopted an approach in which our available SOS resources will be expanded incrementally, using input and suggestions from our own personnel, who are most familiar with the method and its use. We envision as an eventual possibility a system with a full user-friendly interface which will allow even inexperienced computer users to design, and build processes to meet their particular computing needs.

Acknowledgments:

The following individuals (among others) contributed to the success of this project, and their participation is here gratefully acknowledged: Michel Virard, Zbigniew Wojcik, Nguyen Thi Thanh Huong, Ewa Olow, Martin Walker, Wojciech M. Jaworski and David Morton.

References:

- [1] Bergland, G. D., "A Guided Tour of Program Design Methodologies", IEEE Computer, Oct. 1981, pp. 13-37.
- [2] Caine, S. H. & Gordon, E. K., "PDL - A Tool for Software Design", in P. Freeman & A. I. Wasserman (eds.), Tutorial on Software Design Techniques (3rd edition), IEEE Computer Society, 1980, pp. 380-385.
- [3] Cottrell, L. K. & Workman, D. A., "GRASP: An Interactive Environment for Software Development and Maintenance", Database 11:3 (1980), pp. 84-87.
- [4] Date, C. J., An Introduction to Database Systems. Addison-Wesley, London, 1977.
- [5] Falla, M. E., "The Gamma Software Engineering System", Computer Journal 24: 3 (1981), pp. 235-242.
- [6] Fancott, T. & Jaworski, W. M., "Primitive Logic Constructs Considered Harmful in Structured Programs", Canadian Computer Conference, Session 1976 (Mar. 1976), pp. 332-344.
- [7] Ficocelli, L. A., "Problems to Programs: A Humanistic Approach (An Introduction to ABL Methodology)", unpublished master's thesis, Dept. of Computer Science, Concordia University, Montreal, Que., 1982.
- [8] Freeman, P. & Wasserman, A. I. (eds.), Tutorial on Software Design Techniques. IEEE Computer Society, Long Beach, CA, 1980.
- [9] Hamilton, M. & Zeldin, S., "Higher Order Software: A Methodology for Defining Software", IEEE Trans. Soft. Eng. SE-2: 1 (Mar. 1976), pp. 9-32.
- [10] Heacox, H. C., "RDL: A Language for Software Development", ACM SIGPLAN Notices, Dec. 1979, pp. 71-79.
- [11] Hinterberger, H. & Jaworski, W. M., "Controlled Program Design by Use of the ABL Programming Concept", Angewandte Informatik (Applied Informatics), Weisbaden, Germany, July 1981, pp. 302-310.

- [12] IBM Corporation, Data Processing Division, White Plains, NY, "HIPO - A design Aid and Documentation technique", Order no. GC-20-1851, 1974.
- [13] Jackson, M. A., Principles of Program Design. Academic Press, New York, 1975.
- [14] Jaworski, W., MacCuaig, I., Marinelli, T. & Nyisztor, T., "'Executable' Specification for a Large Industrial Process", Proceedings of the 1986 Canadian Conference on Industrial Computer Systems (Montreal, May 28-30, 1986), Canadian Industrial Computer Society, Ottawa, 1986, pp. 60-1 - 60-5.
- [15] Jaworski, W. & Virard, M., "Converting a Software Company to a New Technology", Proceedings of the 1986 Canadian Conference on Industrial Computer Systems (Montreal, May 28-30, 1986), Canadian Industrial Computer Society, Ottawa, 1986, pp. 12-1 - 12-7.
- [16] Kronick, M., "An ABL Software Environment for a Mini Computer", unpublished master's thesis, Dept. of Computer Science, Concordia University, Montreal, Que., 1983.
- [17] Lauber, R. J., "Development Support Systems", IEEE Computer 15: 5 (May 1982), pp. 36-46.
- [18] Lew, A. "On the Emulation of Flowcharts by Decision Tables", Communications of the ACM 25: 12 (Dec. 1982), pp. 895-905.
- [19] Linares, J., "A Comprehensive Support System for Microcode Generation", unpublished master's thesis, Dept. of Computer Science, Concordia University, Montreal, Que., 1982.
- [20] Martin, J., System Design from Provably Correct Constructs. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
- [21] Martin, J. & McClure, C., Structured Techniques for Computing. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.
- [22] Moret, B., "Decision Trees and Diagrams", ACM Computing Surveys 14: 4 (Dec. 1982), pp. 593-623.
- [23] Morgan, A. H., "An Engineering Approach to Problem Analysis", unpublished master's thesis, Dept. of Computer Science, Concordia University, Montreal, Que., 1981.

- [24] Nassi, I. & Shneiderman, B., "Flowchart Techniques for Structured Programming", ACM SIGPLAN Notices 8: 8 (Aug. 1973), pp. 12-26.
- [25] Newell, A. & Simon, H. A., Human Problem Solving. Prentice-Hall, Englewood Cliffs, NJ, 1972.
- [26] Riddle, William E., "An Event-Based Design Methodology Supported by DREAM", in P. Freeman & A. I. Wasserman (eds.), Tutorial on Software Design Techniques (3rd edition), IEEE Computer Society, 1980, pp. 269-283.
- [27] Ross, D. T. & Schoman, K. E., "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, SE-3: 1 (Jan. 1977), pp. 6-15.
- [28] Stevens, W., Myers, G. & Constantine, L., "Structured Design", IBM Systems Journal, 13: 2 (1974), pp. 115-139.
- [29] Teichroew, D. & Hershey, E. A., III, "PSL/ PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering SE-3: 1 (Jan. 1977), pp. 41-48.

Fig. 1

A simple SOS process:

Process Number_Game

	NEXT	EXC
1. Initialize		
A1. Display initial prompt	2	
2. Begin new game		
A2. Get first guess	3	
A3. Exit	0	
3. Evaluate guess		
A4. Guess is too high	3	
A5. Guess is too low	3	
A6. Guess is correct	2	

Actions for Process Number_Game

ACT1. Print "Guess-a-Number Game. To play, type 'Y'"
ACT2. Read character from input.
ACT3. Print "Enter a number between 1 and 10:"
ACT4. Read integer I from input.
ACT5. Print "Goodbye !"
ACT6. Exit
ACT7. Select a random number J in range 1 - 10.
ACT8. Print "Too high. Guess again."
ACT9. Print "Too low. Guess again."
ACT10. Print "You're right ! Play again ? "

Conditions for Process Number_Game

CND1. Input = 'Y'
CND2. I > J
CND3. I = J

Process Number_Game (Detailed Specification)

	NEXT	EXC
1. Initialize		
A1. < > [1, 2]	2	
2. Begin new game		
A2. < 1 > [7, 3, 4]	3	
A3. < -1 > [5, 6]	0	
3. Evaluate guess		
A4. < 2 > [8, 4]	3	
A5. < -2, -3 > [9, 4]	3	
A6. < 3 > [10, 2]		2

S.O.S. DEVELOPMENT

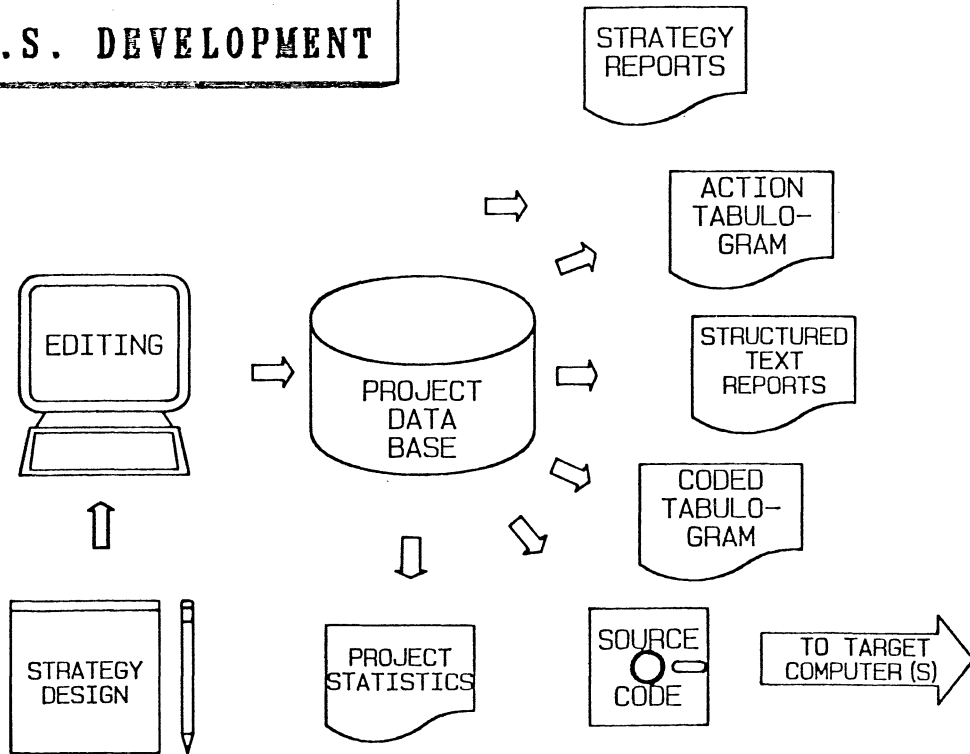


Fig. 3: Project status reports for compiler subsystem development

SOS Project Summary Report	Computer: 10	21-APR-86
Number of Processes:	20	
Number of Strategies:	20	
Number of Environments:	20	
Number of Clusters:	80	
Number of Alternatives:	272	
Number of Actions Defined:	312	
Number of Conditions Defined:	168	
SOS Project Summary Report	Computer: 10	28-APR-86
Number of Processes:	29	
Number of Strategies:	29	
Number of Environments:	29	
Number of Clusters:	114	
Number of Alternatives:	381	
Number of Actions Defined:	436	
Number of Conditions Defined:	227	
SOS Project Summary Report	Computer: 10	05-MAY-86
Number of Processes:	37	
Number of Strategies:	37	
Number of Environments:	37	
Number of Clusters:	149	
Number of Alternatives:	510	
Number of Actions Defined:	652	
Number of Conditions Defined:	308	
SOS Project Summary Report	Computer: 10	26-MAY-86
Number of Processes:	55	
Number of Strategies:	55	
Number of Environments:	55	
Number of Clusters:	239	
Number of Alternatives:	817	
Number of Actions Defined:	1001	
Number of Conditions Defined:	476	

Fig. 4: Example routine in original PDP-9 assembler code:

```

M11, JMS NEXTEC          /MULTIPLICATION FACTOR
    DAC CB+37            /SAVE IT
    JMS NEXTEC          /MEASUREMENT POINT FTR2
    JMS PINSAB          /SAVE PIN FOR TYPING
    JMS TRANSM          /TRANSMIT FTR2 TO STATION
M11..A, LAC (225600)
    JMS TRANSM          /CONNECT ADC TO BUS 8
    LAC (144)
    JMS WAIT            /WAIT 10 MS
    JMS ADCVRD          /GET VC
    DAC M11..S
    LAC (225604)
    JMS TRANSM          /RELEASE ADC FROM BUS 8
    LAC (224600)
    JMS TRANSM          /CONNECT ADC TO BUS 6
    LAC (144)
    JMS WAIT            /WAIT 10 MS
    JMS ADCVRD          /GET VX
    JMS NONEGM          /ZERO ANSWER IF NEGATIVE
    DAC M11..T
    LAC CB+37           /MULTIPLICATION FACTOR
    SMA                 /RS =1 MEG
    JMP .+6             /NO
    LAC CB+24           /VX
    JMS MPYDIV          /MULTIPLY VX BY 1091; DIVIDE BY 1000
    2103                /1091
    1750                /1000
    DAC M11..T          /VX=1.9 VX
    LAC M11..T          /VX OR VX
    CMA
    TAD M11..S
    TAD MASKTB+21
    DAC M11..S          /VC-VX (OR VC-VX)
    SZA                 /PS VOL. = MEAS. VOL.
    JMP .+3             /NO
    LAC (DECIMAL 99999 OCTAL) /YES, ANS. OVERFLOW
    JMP M11..S+1
    LAC CB+37           /MULTIPLICATION FACTOR
    AND (377777)        /REMOVE RS=1 MEG INDICATOR BIT
    DAC .+3
    LAC CB+24           /VX
    JMS MPYDIV          /RS = MUL. FACT. X VX/(VC-VX)
    0                   /MULTIPLICATION FACTOR
M11..S, 0              /VC-VX
    DAC CB+24           /STORE ANSWER
    LAC (224604)
    JMS TRANSM          /RELEASE ADC FROM BUS 6

```


LAC (M11..A+K	
JMS SCCATR	/SAVE ADD. AND TYPE RESULT
M11..B, JMS PINRLS	/ANSWER OK, RELEASE PIN
JMS CHFTR2	/NEXT EC A CONNECT FTR2
JMP ECHAND+1	/NO
JMP M11+3	/YES
M11..T, 0	

Fig. 5: Environment (data flow) for routine of Fig. 4

ENV	ID	I/O	G/L	OBJECT DESCRIPTION
233	160	O	L	ADC..1: XADV RD (CAL) register
233	117	O	L	CB+15: 0 ret.to EXWAIT,10d.-M10+7,11d.-M11+3,12d.-M12+3,14d.-M14+2,16d.-M16..1,20d.-CHKOUT;
233	90	O	L	CB+16: -1 if return address is ECHAND. F1 Flag is sent by ECHAND to EXWAIT, to be tested by CONTIN
233	112	O	L	CB+24: measurement answer storage
233	126	O	L	CB+37: multiplication factor
233	185	O	L	CNTWD: control word for EXEC call; tno : table number for xcall()
233	24	I	L	ERRFLG: Set to one when an error is encountered
233	127	-	L	M11..S: A/D voltage read (Vc) for M11
233	128	-	L	M11..T: A/D voltage read (Vx) for M11
233	900	O	L	Output buffer for clwrite(p) (p=1 if dump immediate, p=0 if dump when buffer is full)
233	6	O	L	TRANA: transmit buffer
233	8	O	L	WAIT.A: buffer for waiting time of AMT5 and AMT6

Fig. 6: Actions and conditions for environment of example process:

ENV	ID	ACTION DESCRIPTION
233	8	Call TRANST (strategy 88, transmit to station)
233	11	Xcall XWAIT (wait call for AMT5 or AMT6) with WAITA
233	253	Call RESULT (strategy 50)
233	259	Store [CB+24] into M11..S
233	264	Store 225600 octal in TRAN.A+0
233	265	5 Store 000144 octal in WAIT.A (10 ms wait)
233	266	Store 225604 octal in TRAN.A+0
233	267	Store 224600 octal in TRAN.A+0
233	268	Clear CB+24
233	269	Store [CB+24] in M11..T
233	270	Multiply M11..T by 1.091
233	271	Store ([M11..S] - [M11..T]) in M11..S
233	272	Store 99999 in CB+24
233	273	Set bit 17 of CB+37 to 0
233	274	Store ([CB+24] x [CB+37] / [M11..S]) in CB+24
233	275	Store 224604 octal in TRAN.A+0
233	276	Set CB+15 equal to 11 decimal
233	474	Set [CB+16] equal to 61 decimal (return address is M11..B)
233	482	Call M11..B (strategy 84)
233	522	Xcall XADVRD with ADC..1
233	523	Store bits 17 and 0-11 of ADC..1 into the same bits of CB+24
233	526	Convert the result of measurement in CB24_ANS
ENV	ID	CONDITION DESCRIPTION
233	86	ERRFLG = 0 (no error)
233	121	[CB+24] < 0
233	123	Bit 17 of CB+37 is 1
233	124	[M11..S] = 0 (PS2 voltage = measured voltage)

Fig. 7 : Strategy (control flow) listing for routine of Fig. 4

SYSLOG report

CF01

19-MAR-86

PROGRAM:33 Strategy for AMT6 resistance measurement

```
1  AMT6 resistance measurement
   A1  Start checking                      --> 2
2  Verify polarity of A/D voltage read (Vx)
   A2  Negative voltage                   --> 3
   A3  Not negative voltage               --> 3
3  Verify multiplication factor
   A4  Resistance Rs is in 1 meg range    --> 4
   A5  Otherwise                         --> 4
4  Verify A/D voltages read
   A6  PS2 voltage is equal to measured voltage --> 5
   A7  Otherwise                         --> 5
5  Verify result
   A8  Result ok                         --> 0
   A9  Result not ok                     --> 0
```

Fig. 8: Coded environment for example process:

ENV	ID	OBJECT CODE
233	160	extern long ADC1
233	117	extern int CB15_REP
233	90	extern int CB16_RET
233	112	extern long CB24_ANS
233	126	extern long CB37_MUL
233	185	extern int CNTWD; extern int tno
233	24	extern int ERRFLG
233	127	long M11S_ADV
233	128	long M11T
233	900	extern char OUTBUF[80]
233	6	extern long TRANA[2]
233	8	extern long WAITA

ENV	ID	ACTION CODE
233	8	TRANST()
233	11	CNTWD = 03000; tno = 1; xcall()
233	253	RESULT()
233	259	M11S_ADV = CB24_ANS
233	264	TRANA[0] = 0225600L
233	265	WAITA = 0144L
233	266	TRANA[0] = 0225604L
233	267	TRANA[0] = 0224600L
233	268	CB24_ANS = 0L
233	269	M11T = CB24_ANS
233	270	M11T = M11T * 1091L / 1000L
233	271	M11S_ADV = M11S_ADV - M11T
233	272	CB24_ANS = 99999L
233	273	CB37_MUL = CB37_MUL & 0377777L
233	274	CB24_ANS = CB24_ANS * CB37_MUL / M11S_ADV
233	275	TRANA[0] = 0224604L
233	276	CB15_REP = 11
233	474	CB16_RET = 61
233	482	M11B()
233	522	CNTWD = 01000; tno = 8; xcall()
233	523	CB24_ANS = ADC1 & 0407777L
233	526	cnvflt ()

ENV	ID	CONDITION CODE
233	86	ERRFLG == 0
233	121	(CB24_ANS & 0400000L) == 0400000L
233	123	(CB37_MUL & 0400000L) == 0400000L
233	124	M11S_ADV == 0L

Fig. 9: Sample of generated source code in C for example process

```
hpc,1,"P033, M11 handler                                Date: 13-MAR-86";
#include <stdio.h>
m11 ()
{
extern long TRANA[2] ;
extern long WAITA ;
extern int ERRFLG ;
extern int CB16_RET ;
extern long CB24_ANS ;
extern int CB15_REP ;
extern long CB37_MUL ;
long M11S_ADV ;
long M11T ;
extern long ADC1 ;
extern int CNTWD; extern int tno ;
extern char OUTBUF[80] ;
fprintf (stderr,"PRC # 033\n");
1001:
{
fprintf (stderr, "PRC 033 ALT # 1 \n");
TRANA[0] = 0225600L ;
TRANST() ;
WAITA = 0144L ;
CNTWD = 03000; tno = 1; xcall() ;
CB24_ANS = 0L ;
CNTWD = 01000; tno = 8; xcall() ;
CB24_ANS = ADC1 & 0407777L ;
```

DESIGN CONSIDERATIONS FOR FOURTH GENERATION LANGUAGE DEVELOPMENT

JONATHAN C. FRENCH AND WILLIAM E. GAINES

INDUSTRIAL COMPUTER CORPORATION
6065 BARFIELD ROAD, SUITE 114
ATLANTA, GA 30328

INTRODUCTION

The benefits of computer technology are well known and understood by management in industry today. The problem is that today's managers have a difficult time responding to the technological advances and changes occurring in the computer and software industries. Software costs are rising, while the backlog of undeveloped applications continues to increase. Traditional software development, utilizing the corporate DP department, is unsuccessful at meeting the needs of end users. As the amount of information users must deal with increases, timely application solutions are essential to a company's ability to respond to changes in the marketplace. In summary, we need a new way to develop computer applications that relies on the user's knowledge of an application, rather than on programmers who may not know the application as well. This would allow the programming staff to catch up with the backlog, and implement the more complex systems that are beyond the scope of end user computing.

With the advent of software automation tools such as Fourth Generation Languages, computer applications can be developed faster, easier, and at a much lower cost. The goal of a Fourth Generation Language is to automate the software development cycle by allowing the end user to develop systems and solve problems typically requiring the assistance of a programmer. The development responsibility is shifted from the corporate data processing department to the end user, who can develop computer applications without having to understand the complexity of his computer system. End user computing makes sense because no one is better qualified to develop an application than the user who conceived the idea.

Traditional software development uses Third Generation computer languages (3GL), such as Fortran or Cobol, which force the developer to break his problem into minute steps that the computer must perform. Conversely, Fourth Generation Languages typically require a problem definition, and a description of the way to display output, allowing the system to define the step-by-step procedures the computer will use. By automating the software development process, many advantages can be realized, such as reduced development costs and increased user satisfaction with the new applications.

DESIGN OBJECTIVES

Two major design objectives must be considered in developing a Fourth Generation Language (4GL). First, the 4GL should provide an interactive and flexible environment that allows the end user to solve problems without needing the assistance from the programming staff. Second, the language should serve as a "tool" that enables the programmer to develop application specific solutions in a fraction of the time required by conventional programming techniques.

Typically, these design objectives are mutually exclusive, resulting in Fourth Generation Languages that are not capable of serving both the end user and the software professional. In light of this, a 4GL system model utilizing an "open architecture" approach will be presented. The open architecture approach to 4GL development embraces the need to satisfy both end users and programmers. When using the 4GL, the user will interact with a friendly end user presentation level that uses natural language and facilitates problem definition and solution. The programmer, on the other hand, will interface with the system thru "hooks" into the 4GL internals. At this level the programmer should have access to all features and subsystems, allowing him to use the 4GL as an aid in the software development process.

THE "OPEN ARCHITECTURE" 4GL MODEL

The "open architecture" 4GL model consists of an integration of the four major subsystems listed below:

- 1) User Interface
- 2) Data Acquisition
- 3) Language Processing
- 4) Application Generation

The model is successful only if each subsystem has the access levels described earlier: end user presentation level and programmer hooks.

User Interface Subsystem

The User Interface Subsystem is extremely important for 4GL success as an end user tool. The purpose of the User Interface is to allow the end user to present his problem to the system using natural language. The natural language should be non-procedural, describing *what* is to be done rather than *how* it is done. The "human factor" principles incorporated into this level should be extensive, providing help, tutorials, and ease-of-use functions. For acceptance by end users, the User Interface should be built on a menu-driven forms system which contains a screen painter and an automatic form builder. To aid the programmers, there should be programmatic access to the forms system allowing on-line transaction development without the burden and complexity of having to know escape codes and the operation of different terminals.

Data Acquisition Subsystem

The purpose of the Data Acquisition Subsystem is to manage database access within the 4GL, while shielding the end user from DBMS complexities. The Data Acquisition Subsystem is based on a flexible data dictionary which allows both the user and programmer to describe how the 4GL can access data items available to end users. As data items are entered into the dictionary, this subsystem maintains information about the relationships between available data items. This capability permits relational database access, even if the underlying files are not a part of a relational DBMS. Also, the Data Acquisition Subsystem must be able to pull together information from separate file systems or database systems, since the end user will not know where the items he is asking for are located.

Language Processing Subsystem

The third component of the 4GL model is the Language Processing Subsystem. This subsystem permits end users to enter natural language sentences. To accept the end user's natural language, the language processor consults a language dictionary to verify language elements and grammatical constructs. If the sentence is accepted, the language processor translates the non-procedural natural language into a command metalanguage. The metalanguage consists of a list of procedural commands which describe how the computer will solve the user's problem. In order to increase the capabilities of the 4GL, a primary design goal of the Language Processing Subsystem is to permit programmers to expand on it by adding new language. Also, the end user should be able to customize and personalize the language so that it is easier to use.

Application Generation Subsystem

The final piece of the model is the Application Generation Subsystem. This component of the model is responsible for translating the command metalanguage into executable object code. After the application is generated, the end user should be able to run the application or catalog it for later use. Optionally, the Application Generator should be capable of providing automatic documentation of the application. Also, the Application Generator should provide the means for a programmer to modify the "normal" application generation process. Such a feature would, for example, permit programmers to invoke modules of their own.

DESIGN DECISIONS REQUIRED BY THE MODEL

Two major design decisions have been made which influence the development of a 4GL under the "open architecture" model.

First, the system should not be internal to its own database system. Instead, it should be able to access any database or data file system merely by "acquiring" a description of the access method to the database or data file. This decision allows the 4GL to use existing data files without requiring re-entry of all existing data elements into the system.

The second design decision is that the Application Generator should write high level language source code in a variety of different languages, such as Fortran, C, and Pascal. The code produced should be well documented and commented so that a programmer can easily customize or modify it. Also, the 4GL should encourage programmer enhancements by allowing language invoked functions and calculated data items.

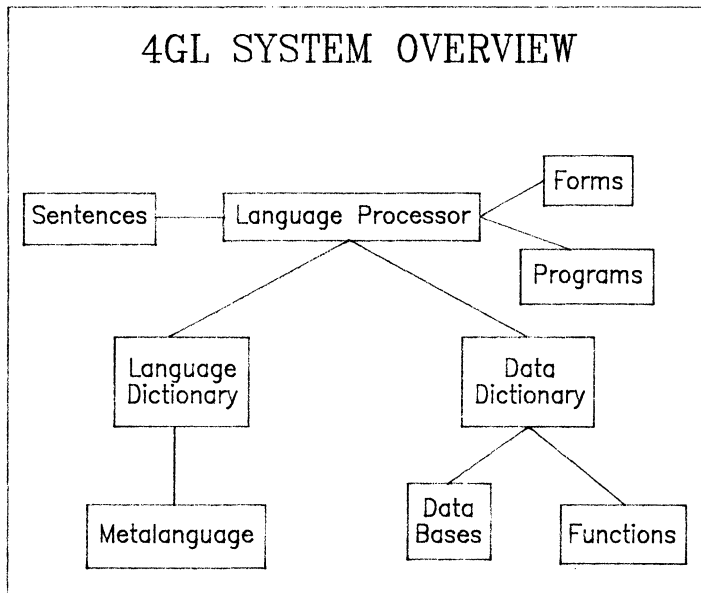
MODEL DATAFLOW

Based on the system model and design decisions presented earlier, the 4GL we have described must translate natural language into high level source code. In order to do this, the Language Processor relates sentences into a set of metalanguage commands which form an outline for the software solution to the problem. The command language is essentially a super high level language in that one metalanguage command corresponds to many 3GL source code lines. To write the code necessary to run the application, the Application Generator expands the metalanguage list into a source code "shell", and fills in the application specifics based on the data items used. The 4GL system must then compile and link the newly created program so that the end user may execute it. In summary, the 4GL system converts natural language to metalanguage, and then metalanguage into 3GL source code.

AN IMPLEMENTATION OF A FOURTH GENERATION LANGUAGE

We will now describe a Fourth Generation Language (4GL) system that conforms to the model previously described. Our system for a 4GL that is easy for the end user to use, as well as versatile enough for a programmer to use, is divided into four subsystems. They are the User Interface Subsystem, the Data Acquisition Subsystem, the Language Processing Subsystem, and the Application Generation Subsystem.

The 4GL System Overview diagram below encompasses the four subsystems that we will be discussing in the remainder of this paper. You will want to refer back to this figure as you read through this section.



The **User Interface Subsystem** is a menu-driven interface that makes it easy for an end user to create solutions to a problem. These solutions can range from entering data in a database to generating a report of data.

The **Data Acquisition Subsystem** allows the database administrator or programmer to enter the structure of the data as well as the list of data items that an end user may use in the User Interface Subsystem. This subsystem must contain enough protection to prevent the user from accessing restricted data.

The **Language Processing Subsystem** is the natural language interface that the end user and programmer use to generate a solution to their specific problem. The natural language interface allows the user to enter English-like sentences to describe the problem that needs to be solved.

The **Application Generation Subsystem** builds an executable program that will run on the system and generate the solution to the user's problem. Optionally, it generates actual Third Generation Language (3GL) source code that a programmer can later modify.

USER INTERFACE SUBSYSTEM

The User Interface Subsystem is the part of the 4GL with which the end user actually interfaces. It allows the end user to solve a problem by entering natural language sentences. It also has plenty of on-line help for the user whenever the next step or operation is not obvious.

The User Interface also contains the following human factors.

- o The means of establishing contact and signing on the 4GL is simple, natural, and obvious.
- o The user needs to know very little about the system itself to get started.
- o The user does not have to remember mnemonics or alien syntax.
- o The user always knows what he needs to do next.
- o All error messages are self explanatory.
- o There is full use of a data dictionary, directory, or encyclopedia.
- o Although the technique for achieving user friendliness may seem slow or oversimplified, an expert mode is provided which allows the user to use a faster, more direct technique when he becomes experienced.
- o The software is self-teaching, with good quality on-line help that can be invoked at any point while building the solution to the user's problem.

The User Interface also maintains database integrity via a protection scheme. In addition, it is capable of producing production quality reports, forms, and graphs.

The User Interface is menu-driven and performs four functions:

- 1) It defines a new application program.
- 2) It edits an existing application program.

- 3) It executes previously defined application programs.
- 4) It deletes existing application programs.

Applications generated by the User Interface can be defined as either temporary or permanent. The user may generate a temporary application if it is to be used only once. This type of application can include reports or inquiries of a database as well as one time data entry functions. When the user exits the 4GL system, the system will automatically remove any temporary applications that the user defines. The user may generate a permanent application if it is to be used more than once. When the user leaves the 4GL system, all permanent applications will be saved.

The 4GL system is also capable of generating two types of application programs. The first is a program that uses a database. This type of application program is used for data entry into a database, or for generating reports or graphs from existing data in a database. The second kind of program is a menu. The menu program is used to give the end user a menu-driven interface to his application programs. Database programs or other menu programs can be scheduled from a menu.

To prevent the user from accessing and changing database information that he should not be using, two methods of protection are provided. Both of these security methods are used by the database administrator to control the information that the user may access.

The first method of protection is the implementation of read/write access permission for a specific database item or piece of data. By using the read/write access flag, the database administrator can control which pieces of data can be accessed or written in the database.

The second method of protection is the use of data relationships to limit the access of users. A data relationship specifies the data items (or pieces of data) that a particular end user or groups of end users may use. When the end user logs into the 4GL system, he must select the data relationship that he wishes to access. This immediately limits the database items that he may use.

The define and edit functions of the User Interface are very similar. They allow the user to enter natural language sentences to describe the problem that he wishes to solve.

The sentence entry part of the 4GL system is very easy for the end user to use. At the bottom of the sentence entry form, choices for the next possible word are listed. Once the end user chooses one of the words, the 4GL system automatically updates the list for the next set of possible words. This type of sentence entry is designed for the novice user. Once he becomes more familiar with the natural language, he can enter a sentence in its entirety rather than a word at a time.

Once the user has described his problem, he will generate a form or set of forms using the screen painter that describes the layout of the report or data entry form. This is performed by moving the cursor around on the terminal screen and pressing a function key when the cursor is correctly positioned for the current data item.

After a program and forms have been generated for a data entry or report application, the user has the option of executing it immediately or at a later date. The execute function will remember the applications the end user has generated during the current 4GL session, and allow him to easily execute any of them. The user may also execute applications generated during earlier sessions.

The last function that the User Interface performs is the deletion of existing programs (applications). It will only let the user delete programs that were generated with the 4GL system.

The User Interface allows the programmer to play more of an assistance role in developing end user applications. Instead of having to write lengthy software programs, the programmer can concentrate on enhancing existing applications and programs. He may occasionally have to assist the end user to implement solutions that cannot be generated with the 4GL. He may implement these solutions by using calculated data items or by providing his own user-defined functions. As a last resort, he may also modify the program source code generated by the 4GL system.

The User Interface helps the end user in several ways. He can develop reports, data entry forms, and graphs without having to wait for a programmer to catch up with the backlog --- a situation that sometimes makes the solution to a problem come too late. The end user can also use the 4GL system to prototype a complete application system with a few sample programs without having to rely on a programmer.

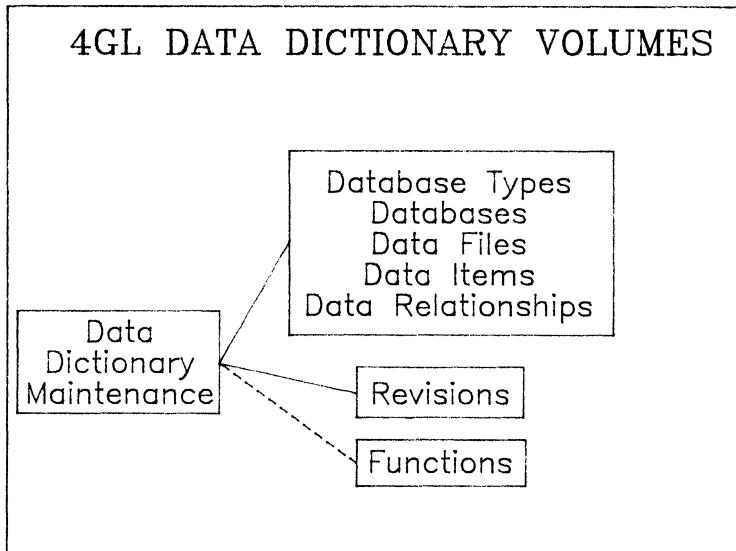
DATA ACQUISITION SUBSYSTEM

The Data Acquisition Subsystem allows the database administrator or programmer to enter the information about all of the databases and files that are available on the system where the 4GL will be used. It is based on a data dictionary system that makes a relational database system out of databases that may or may not be relational.

The data dictionary is divided into several volumes that allow the administrator to specify how the data in the databases is to be accessed. The volumes that compose the data dictionary are as follows:

- 1) Database Types Volume
- 2) Database Volume
- 3) Data File Volume
- 4) Data Item Volume
- 5) Data Relationship Volume
- 6) Revisions List

The figure below illustrates the Data Dictionary Volumes and their relationship to the Data Dictionary Maintenance function.



Data Base Types Volume

The database types volume contains entries to indicate the types of databases that exist on the user's system. Examples of these are flat file databases, IMAGE databases, and any of the relational DBMS's that are available. For each database type there is a set of generic database access routines that will access the type of database specified. These routines are provided with the fourth generation product, but can be added to by a programmer when a new type of database is needed for which there are no supported generic access routines.

Database Volume

Once the database types volume has been given the information about all of the database types, information may be entered in the database volume. The database volume contains the names of all of the databases on the user's system. For each database, the administrator must enter a database reference name, where to find the database, and a database type. By entering the database type, the administrator has given the 4GL system the information needed to access the specified database.

Data File Volume

After the administrator has entered the information about the databases on the system, he may enter the information about the files. The data files volume contains the information necessary to access the different files on the system. For each file entry, the administrator must enter a data file reference for the file, the file name, and a database reference. By specifying the database reference, the administrator has linked the data file to a database on the system.

Data Item Volume

After the administrator has entered the information about a data file, he enters information about each individual piece of data (data item) that will be accessed. Two types of data items are supported. The first type is one that resides somewhere in a data file. The second type is one which is calculated and does not reside in a file.

File Resident Data Item

For each individual data item that resides in a file, the administrator enters a data item reference name, the actual data item name, a "picture" of how the item will look on a form or report, the highest and lowest acceptable input value for numbers, and a list of all of the file references where the data item resides. The data item "picture" may need further explanation. This is a COBOL-like picture which specifies how the data item will look on a form or report. For example, the picture that might be used for a social security number data item is as follows:

NNN-NN-NNNN

The "N" in the picture indicates that the data item is a numeric data item with 9 data positions (9 ASCII characters). The "-" represents a character that will be placed on the form or report to divide pieces of the data item. Almost any character (even a space) can be used to divide a picture.

The last step needed to set up the data items that reside in files is the entry of the data file references that each data item resides in. Along with the data file reference, a read/write indicator will be specified, and whether the data item is to be a key path in the specified file. The read/write indicator helps limit access to a data item if necessary. The key path helps the 4GL make a better choice at how it should read the files that are required.

Calculated Data Item

The calculated data item requires basically the same information as the data item that resides in a file. The difference is that an equation is given for it instead of a list of data file references. The equation may contain other previously defined data items, arithmetic operators, or **functions**. The previously defined data items may be either data items in files or other calculated data items. The functions that may be used in the calculations are either provided by the 4GL or may be added by the programming staff.

By allowing a programmer to enhance the end user language via calculated data items and functions, site-specific enhancements can be made to the 4GL. Using functions to make enhancements keeps the programmer from modifying 4GL generated source code that cannot be reproduced when the end user needs changes made. When the enhancements are put in a function, the end user can change his problem without worrying about whether the site-specific changes will be remembered --- they always will.

Data Relationship Volume

Once the database administrator has entered the information about data items, he may set up the data relationships that will be used to control user access to individual data items. Inside the data relationships volume, the administrator gives a data relationship reference name and then a complete list of all data items that the user of the data relationship can use. An end user using a particular data relationship, then, does not have to worry about which files are required for what data items. Only if a data item resides in more than one file will the end user be required to select the file of his choice. The lack of files helps break down the access barriers and allows the database administrator to make a relational database out of databases or files that may not be relational.

Revisions List

One of the last parts of the Data Acquisition Subsystem, and certainly one of the most important, is the revisions list. The revisions list is a file which contains a list of all of the changes that have been made in the data dictionary (excluding new entries in any of the data dictionary volumes). The revisions list is very useful because it recreates all of the programs that need to be changed due to a data dictionary change. One reason that a program should be recreated is that the byte position of a data item has changed in one of the data files. Another reason might be that a data file has been changed to have a shorter or longer record length. The program recreation process is either performed automatically at certain intervals, or on demand by the user.

The Data Acquisition Subsystem allows the programmer to focus more on the application that he is providing rather than the implementation of the application. This is because all of the tedious parts of programming have been removed. For example, opening and closing data files, formatting the report, and accessing the database are functions that the 4GL can automate for the programmer.

The Data Acquisition Subsystem helps the database administrator because he does not have to specify how to access data in different files. This eliminates the problem of some 4GL systems that require the database administrator to specify the links between the different data files that will be used by the end user. This 4GL system will automatically generate that information as it is needed.

LANGUAGE PROCESSING SUBSYSTEM

The Language Processing Subsystem is the natural language interface that the end user or programmer uses to specify the solution to his problem. The natural language interface is based on English-structured sentences that the end user enters into the 4GL system.

Consider the following example.

**PRINT REPORT OF MINOR REPAIRS FOR TERMINALS OF TYPE
"TOUCHSCREEN".**

The breakdown of this sentence does not adhere to strict English grammar, but the natural language interface interprets it as follows:

VERB: print report
NOUN: repairs
DESCRIBING ADJECTIVE: minor
LIMITING ADJECTIVE: terminals of type
CONJUNCTION: for
NULL WORD: of
CONSTANT: "TOUCHSCREEN"

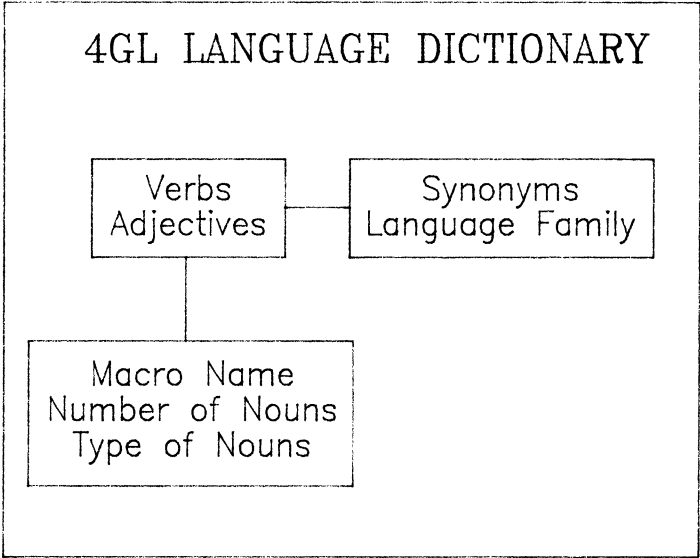
The natural language interface describes the individual pieces as follows:

- 1) **Verb** - The action statement in a sentence. In our natural language, the verb must always be the first word in a sentence.
- 2) **Noun** - The object of the verb (the language allows more than one side-by-side).
- 3) **Describing Adjective** - Precedes the nouns and describes the scope of the nouns.
- 4) **Limiting Adjective** - This adjective phrase imposes some limiting constraints on the sentence. Limiting adjectives follow the noun.
- 5) **Conjunction** - A joiner of limiting adjectives.
- 6) **Null Word** - A word that is ignored by the natural language. Null words help to make the language more natural to the end user.
- 7) **Constant** - Fills the place of a data item with an actual value.

Please note that the term adjective is used very loosely in our natural language description. The definition implies a describer or modifier. Further, no distinction is made between English language adverbs and adjectives since both help describe some action in the natural language.

Also note that when we describe a word in the natural language, it can be a single word or a phrase of words depending on how the language is entered. In the example sentence, "PRINT REPORT" is considered one word in the natural language. A discussion about entering the natural language will come later.

Now that the components of the language have been described, we can show how the natural language is actually pieced together. Another data dictionary (called the language dictionary) is used to hold the natural language for the 4GL system. This permits the easy addition of natural language. Additions can be made by the 4GL supplier as well as the programming staff of the company using it. Additions to the language that have been added by a firm using the 4GL are always used before the language supplied with the product. This allows the meaning of the supplied language to be changed if necessary.



There is an entry in the language dictionary for each word that is available to the end user. Remember, a word can be a phrase. Each word may also have a number of synonyms. For example, one end user may prefer to enter the words "PRINT REPORT" to get his report, while another user may prefer "SHOW ME". Each synonym has a language flag associated with it so that if a user prefers to use language other than English, he may have it specified in his user accounting information for logging onto the 4GL system, and the 4GL system will automatically only give him choices in the language specified.

For each word in the language dictionary, there is also a macro directive (metalanguage command) associated with it. The 4GL system uses the macro directive to indicate what type of programming language source code will be used to perform the function necessary for the word. The macros will be explained in more detail when the Application Generation Subsystem is described.

Each word in the dictionary may specify that a noun or nouns follow it. These nouns can either be data items from the language dictionary, or constants. The programmer that sets up the language has the choice of whether data items, constants, or both are allowed. An example of this is:

SELECT USERS

SELECT is the word (verb) in the language dictionary and USERS is a noun (data item) in the data dictionary. In the language dictionary, the word SELECT would specify that it allows one noun to follow it that is a data item.

Another example is:

DISPLAY USER "211-01-2322"

In this example, DISPLAY is the word (verb) in the language dictionary, USER is a noun (data item) in the data dictionary, and "211-01-2322" is a constant. In the language dictionary, the word DISPLAY would specify that it allows two nouns to follow it, one of which is a data item, and the other a constant.

For each word in the language dictionary, there is also a list of describing adjectives that the natural language will accept prior to the noun in a sentence. These describing adjectives have an entry in the language dictionary just like any other word. An example using a describing adjective is:

PRINT USER SELECTED TOYS.

In this sentence, PRINT is the verb, USER SELECTED is the describing adjective, TOYS is the noun. In this example, USER SELECTED refers to the process of querying the end user for a TOY prior to printing all of the data about it.

With each word in the language dictionary, a set of limiting adjectives may also be specified. The limiting adjectives are used to limit the scope of the verb. They also tie the words in a sentence together. When the natural language sentence processor is evaluating a word in the sentence, it knows what the next word may be by checking the limiting adjectives. When it finds a correct limiting adjective, it proceeds through the sentence and begins processing the limiting adjective as if it were the beginning of the sentence. An example of using a limiting adjective is:

PRINT TOYS WITH MANUFACTURE DATE GREATER THAN "12-01-86".

In this sentence, WITH and GREATER THAN are limiting adjectives. When the language processing system is interpreting the sentence, it divides the sentence into the following pieces.

- 1) PRINT TOYS
- 2) WITH MANUFACTURE DATE
- 3) GREATER THAN "12-01-86"

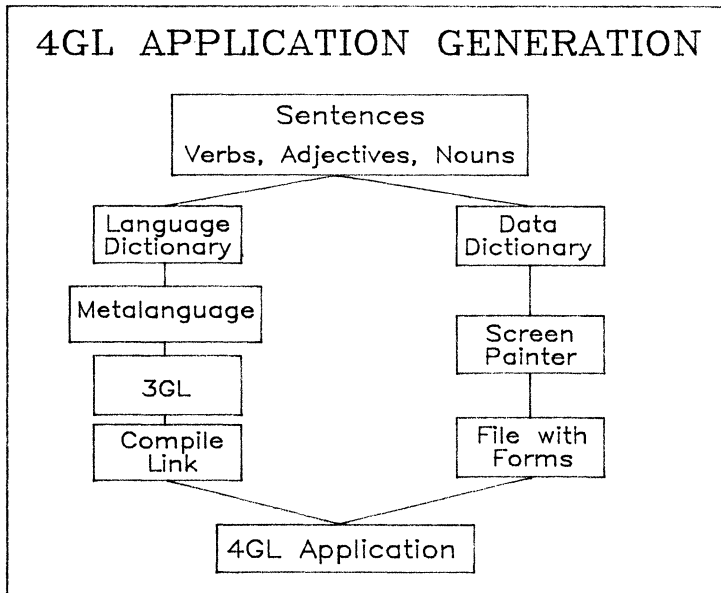
The last piece of information that can be entered in the language dictionary is used only for verbs (the action statement and first word in a natural language sentence). It is a list of all of the verbs that may start the sentence following the current one. This allows the language to proceed in a manner that keeps the end user from typing in sentences that are not related.

This method of language construction helps the programmer by redefining part of his job. Since each word in a sentence is related to a macro, it is the programmer's responsibility to code these macros. When the programmer writes these software macros, he is solving a particular problem in general without regards to the database or data file that he will need to access. While it may take twice as long to implement software in this form, the resulting piece of software may be used over and over again by the 4GL system without any modification. The source code has already been debugged once and should never have to be done again. In the long run, the programmer spends less time on user needs.

APPLICATION GENERATION SUBSYSTEM

The Application Generation Subsystem is the part of the 4GL system that actually converts natural language sentences to an executable software program. Three translations occur during this process. First, the natural language sentences are converted to an internal metalanguage. Second, the internal metalanguage is converted to a third generation software language. Finally, the computer system's compiler and linker are used to convert the 3GL software code to an executable program on the end user's computer.

The diagram below illustrates the relationship of these translations.



The first phase of the translation is to convert natural language sentences to a series of macro directives. These directives are obtained by the language processor when it scans the sentences that have been entered by the end user. Since each word in the language dictionary corresponds to a macro, the macro name is used along with any data items that the end user has specified to generate a macro directive. The following example illustrates how a sentence is translated into macro language.

SELECT TOYS WITH MANUFACTURE DATE EARLIER THAN "12-01-86".

The macro directives generated for the sentence would be as follows:

```
SELECT,TOY
IF,MANUFACTUREDATE
LESS-THAN,"12-01-86"
THEN
```

The nouns have been compressed to remove blanks from their name. Also, the macro directives for some of the language are different from the natural language. For example, the natural language word WITH generates a macro directive named IF.

In order for the next phase of translation to be performed, a file must exist with 3GL source code for each macro directive that has been used. The 3GL source code that is in the macros are well commented and use standard portable code. The code also uses generic database access calls to allow a programmer to add a new kind of database at a later time. Database and file specifics are always gathered at run time to allow for data dictionary changes.

The last phase of translation is from 3GL code to an executable program. This is performed by scheduling the system's compiler and linker if necessary.

One of the nice things about this type of three phase translation is that the output of the first two phases is optionally left for the programmer to modify. He may modify the macro directives from the first phase of translation and change the form of the resulting program. He may also modify the commented 3GL source code that is output from the second phase of translation.

Once again, the use of the macros allows the programmer to develop code once using generic database access calls that can be used over and over again without any enhancements. It also frees him to further extend the natural language with other macros or write 3GL source code that cannot be generated from a 4GL system.

CONCLUSION

The 4GL model we have presented can serve both the end user and the programmer. Our 4GL implementation provides a powerful, easy to use applications development facility for end users. This 4GL product is capable of reducing DP costs, while increasing end user satisfaction levels as their software needs are met in a timely manner. This implementation serves the programmer equally well as an aid to development, prototyping, and documentation. It even increases programmer job satisfaction by eliminating the tedious development required to generate end user reports and on-line applications.

In future times, software development will rely heavily on software automation with the 4GL. If implemented as we have described, not only can the 4GL redefine the programmer's job by placing him in a role that better utilizes his technical skills, but also the end user can become a central part of the application development process.

A Modular Integration of Factory Cells

Robert C. Combs
C & L Systems
1250 E. Ridgewood Ave
Ridgewood, NJ 07450

Introduction

The need for programmers continues to rise and the supply of available talent continues to fall behind that growing need. This coupled with the increasing expense of software development has created a thirst for standardized software. But as the problem of "customized" applications continues to wane, there are solutions. One approach to this dilemma is to utilize standard modules which can be arranged in many different configurations. With a modular approach, a large amount of the application can be implemented with standard packages, using custom programs to complete the total system.

This paper will discuss a modular approach to factory cell data acquisition and various implementations that can and have been achieved with this approach. First the factory cell will be defined with identification of the goals desired from its automation. The idea of a modular design approach will be presented, detailing the various modules and their functional abilities. The utility of these modules will be examined by exploring a few actual and possible implementations.

1. Definition of Factory Cell

Many are already familiar with the term "factory cell". To others, it might be known as a workstation, unit, pilot unit, test cell, or test stand. For the purposes of our discussions, these are all equivalent.

The factory cell is a definable piece of automation or machinery that can be treated as a separate entity. There are signals coming from the cell, such as temperatures, pressures, flow rates, voltages, power usage, etc. Signals may also be going to the factory cell for control of some of these parameters.

The cell is usually self-contained unto itself and is responsible for performing some specific task or function. Typical functions might include part manufacturing, mixing/processing measuring, evaluating, or testing.

Factory cell automation is the first level of Computer Integrated Manufacturing (CIM). Without well structured cell control, an integrated

facility cannot be implemented with any measure of success. Therefore, the factory cell automation should be well planned.

2. Identification of Cell Goals

Before any design can be effective, it must have clearly defined goals. Without clear goals, any design will do, but never with any measurable success.

The cell's purpose may have any of several different purposes for being. Its mission may be to produce a product, to perform quality control checks, to test a new production technique, or to run research experiments. Determining the cell's mission will help define some of the other requirements.

Questions that must be answered before beginning the design are:

What types of signals must be acquired? Any analytical results?

What are the control requirements?

What types of data storage/archiving are needed?

Is there a host computer? How is data sent to it?

These types of questions will help define the goals of the factory cell and its data needs.

3. Modular Design

A modular design is a design that makes use of smaller building blocks to quickly implement a variety of data acquisition and control strategies. The advantages of using standard modules include fast system implementations, reduced maintenance problems, standard operation techniques, and compatible data structures. The standard modules or building blocks that will be used in this paper are identified below.

DATA ACQUISITION

The Data Acquisition module scans the signals and converts all values to engineering units. It performs alarm checking, processes readings, and trends signals.

CALCULATION & CONTROL

The Calculation & Control module provides the ability to perform computations with the measured signals and to execute standard PID control functions. Outputs are generated by this module which are used to achieve the desired continuous control of the cell.

HISTORIAN

The Historian module stores data and events into files for later access. Generally these files are circular so that data is automatically thrown away after some period of time. There can be data compression techniques used within this module to conserve disc space.

DATA BASE INTERFACE

Long term archiving of data is accomplished by storing the historical data into a data base using a Data Base Interface module. This module may actually be used to transmit the information to another system rather than storing in a data base.

GRAPHICS

A Graphics module allows color graphic displays of the data to be constructed. The purpose of this module is to allow custom user interfaces to be easily and efficiently generated. These displays are periodically updated in real-time when used.

BATCH

The Batch module provides the ability to define the sequential steps through which the phases of the cell are to be processed. This sequencing includes either event driven, time driven, or a combination of the two to achieve total flexibility.

LABORATORY ACQUISITION

Results from the analytical equipment, whether process or analytical GCs, etc., are processed into the data base by the Lab Acquisition module. The results are time correlated into the data base with the data from the Data Acquisition module so that a unified report can be generated from all pertinent data.

These modules are interconnected to solve various cell control problems.

It is important to view cell automation design from a layered aspect. The initial layer, the most basic layer, is the data acquisition.

The next layer is the continuous control. Continuous control takes the values from the data acquisition layer and computes the control outputs. On-line computations are also done in this layer, as are adaptive control algorithms.

The sequential control layer sits over the continuous layer. Sequential control monitors the acquired and computed data and makes changes in continuous control parameters based on the sequential logic.

The management layer determines what tasks the sequential layer should be executing and downloads appropriate orders. This layer would typically be used to select the product to be manufactured or the type of testing to perform. This layer is not discussed in this paper.

4. Case Examples

To illustrate the use of this modularity, let's examine a couple of different configurations.

CASE I

The first case that should be examined, is a basic design that really only requires one or two modules. In this application, a cell is to be monitored, looking for conditions out of limits. The system's sole purpose is to watch for production problems and hazardous operational conditions.

This design consists only of the Data Acquisition module. The Data Acquisition module performs the measurement scanning and the alarm processing. An annunciator can be used so that alarms trip an audible warning. In the case of special shutdown errors, the alarms can schedule a user written program to stop the cell. The Graphics module can be used to augment the operator displays, showing error conditions in yellow or red colors for quick problem assessment.

CASE II

This case is a straight forward system that is to measure and control a process, and send the results to another host computer. The local operators must be able to review the current and prior shift's data; the last 16 hours worth of data. The data to be sent to the host computer is to be placed into a file by shift. The host will read the shift file over the link at the require time.

In this design, only three of the modules are needed. The Data Acquisition and Calculation & Control modules will perform the fundamental operation of the process. The Historian will take the data and place it into files, changing the filename at the end of each shift. There are three different filenames used; one for each shift. As the Historian steps to the next file, the host computer will fetch the previous file. Naturally there is a preset time delay between when a shift ends and when the host comes looking for that file. This ensures that the data file is complete.

CASE III

In this case, the system must not only gather cell data and perform control, but also pick up analytical results and report on the total aggregate data. The system is wholly self-contained and must therefore archive the data in a data base for future reports and comparisons.

This design utilizes most of the modules. The Data Acquisition and Calculation & Control modules generate the data which the Historian places in a circular file. The Data Base Interface module places that data into the IMAGE data base for permanent storage. Analytical results from gas chromatographs, etc., are picked up by the Laboratory Acquisition module and placed in user files. The results might normally be stored directly into the data base, but in this system the user needs to perform custom processing of the results before saving them. Once the user's processing is complete, the user file of results are stored into the IMAGE data base. The stored readings are time correlated with the measured data, using the injection time of the sample. In this way the analytical results will reflect correctly with the process state.

The Report module allows reports from the combined data base information, and optionally a graphic report may be obtained.

One little incidental the system has is a data base Cleanser module. This module can be configured to either clean specific data out of the data base upon interactive operator commands, or to periodically purge old data.

5. Summary

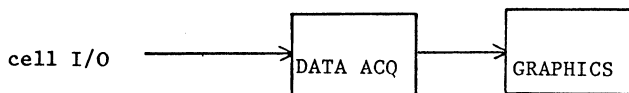
As has been shown, the modular approach to data acquisition system design can affectively be applied. The results are solid systems with a conformity of software that makes maintenance of the software viable.

The case examples, although disguised, are real proven implementations of systems that my company has installed. The modular concept is field tested and proven.

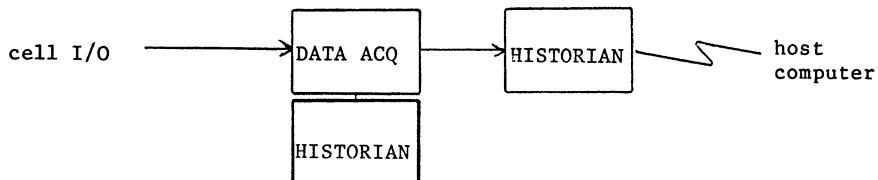
The advantages of standardized modules, to reiterate, are

- * fast system implementations
- * reduced maintenance problems
- * standard operation techniques
- * compatible data structures

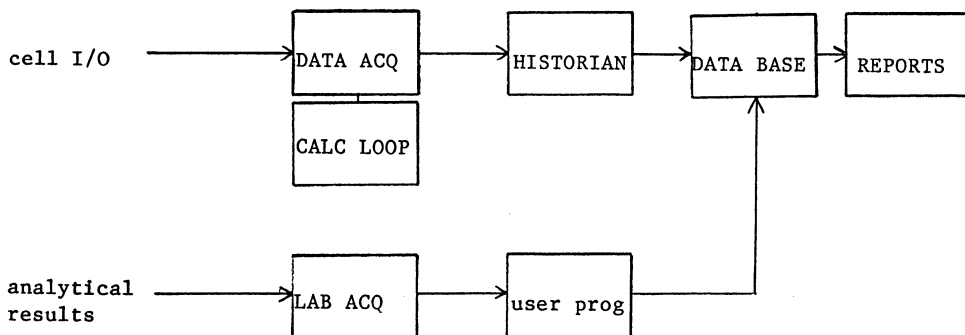
These advantages result in labor and time savings, reduced costs for maintenance, easy training and shifting of personnel, and easier, quicker integration of cell control into the manufacturing network. And from management's point of view, that is what its all about.



CASE I



CASE II



CASE III

CREATING CUSTOMIZED QUALITY CONTROL CHARTS

Cal Bonine
Statware, Inc.
P.O. Box 510881
Salt Lake City, UT 84151

Introduction

Since many production facilities are turning to automation whenever practical, quality control (QC) programs must also evolve to keep pace with the dynamic forms and quantity of data as well as the changing types of analyses. In contrast to a program that only accepts data in one form and performs a canned analysis, a flexible, customizable program offers many benefits.

A flexible analysis system should:

- Be easy to use
- Allow different forms of data
- Provide extensive data management abilities
- Be easily customizable
- Allow the user to extend the system

With these goals in mind, the packages STAT80 and GRAFIT* have joined forces to provide HP 1000 users with a complete set of procedures to produce professional, presentation quality QC charts.

In this combination STAT80 provides the user interface, statistical computations and acts as a driver for GRAFIT which provides the graphics output. Since GRAFIT is a separate package, one can interactively customize the QC charts by adding graph annotation and many other advanced functions which GRAFIT supports.

* STAT80, a data analysis system, is a copyright program of Statware, Inc. (801/521-9309).

GRAFIT, a technically-oriented graph generation package, is a copyright program of Graphicus (408/246-9530).

Ease of use

For a program to be useful it must be easy to use.

Since the syntax of all STAT80 commands is the same (including user created procedures, which will be explained later), the program is both easy to use and quick to learn.

The basic syntax is of the form:

```
command-name    variable(s)    /option(s)
```

All "command names" may be viewed by entering HELP COMMANDS or for STAT80's QC charts in particular, HELP QC. The HELP system for a particular command will describe the command and each of its options. "Variable(s)" refers to columns of numbers that are identified as V1 for the first column, V2 for the second column, etc. (Note that a variable may also be labeled and then this label used for reference.) "Options" permit the user to give additional information that is not normally required by the command. Options form one basis for customization and will be shown shortly.

As an example of the ease of use of these charts, the single STAT80 command, XRCHART_GRF ALL, will produce the results displayed in Figure 1 (assuming that the appropriate data are in STAT80's workspace).

\bar{X} and R Charts

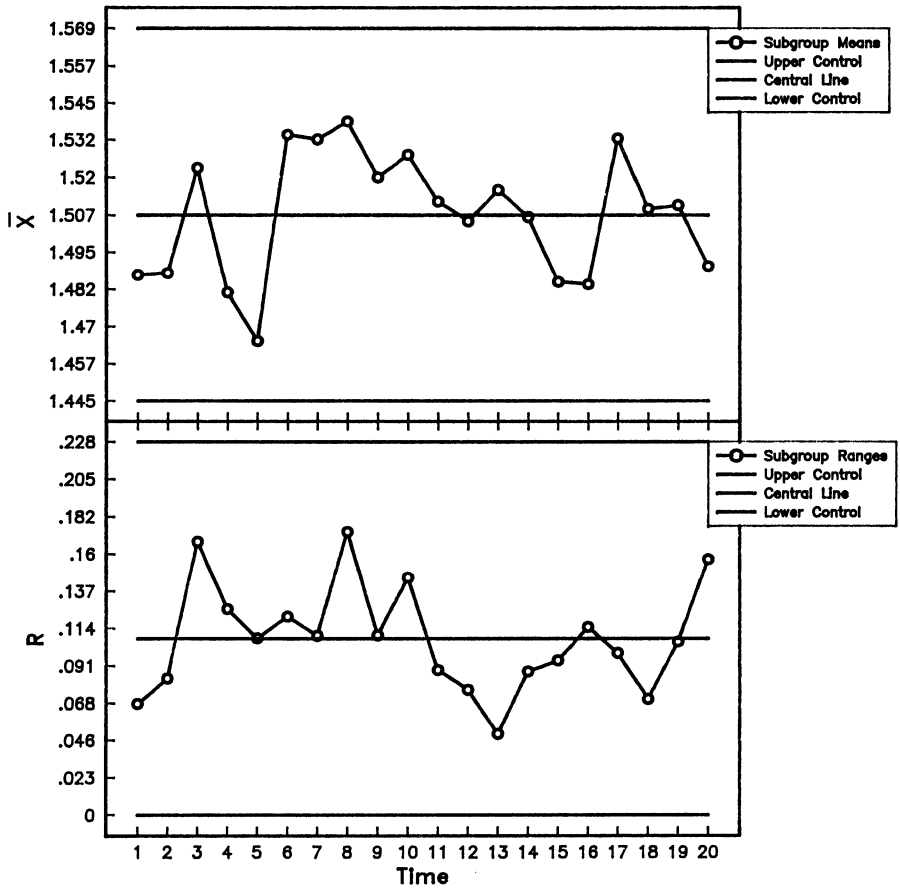


Figure 1

Jun. 26, 1986
13:50:48

To help the user when entering commands, STAT80 provides an in-line help facility which is very useful and easy to use. The in-line help may be used to display the form of the command and also list the options that are available. When the user is entering a command and would like to be reminded of what is required by the command, simply enter a question mark and press the return key.

For example, Figure 2 illustrates the use of in-line help for the command XCHART_GRF. (User input has been underlined.)

STAT80 Analysis System -- Release 3.00 -- 10-Jun-86 14:17:19
Copyright (C) 1985 -- Statware, All Rights Reserved

```
Ready
getfile xrchart.wrk
Retrieving STAT80 file:  xrchart.wrk
Written Mon Jun 9, 1986  09:42:45 AM,  STAT80 version 3.0
Variables V1 to V20 read.
Ready
xchart_grf ?
xchart_grf ? <variable list> Variable(s) containing measured data
Ready
xchart_grf v1 to v20 ?
xchart_grf v1 to v20 ? <variable list> Variable(s) containing measured data
or one of the following option(s):

    /Axis           /Device           /Mstandard         /NOexit
    /NSigma         /Rstandard        /SAVE              /SIZE
    /STddev         /Title
```

Or a RETURN to execute the command.

Ready

Figure 2

Thus, STAT80 is easy to use and very helpful. This is particularly true for the novice, yet since the in-line help is brief and instantaneous, it also serves as a quick reminder to guide the seasoned user whenever necessary.

Flexible Data Formats

A potentially serious drawback for many QC software packages is that they can only handle one form of data. This may present problems since companies often have different types of machines collecting data. Taking this into consideration, STAT80 has the ability to process data in a variety of formats. The examples given below read data from files. STAT80 also offers three methods of entering data interactively.

The following data set, called "wide.data", was collected on 20 variables (subgroups) each with 5 observations. The first record contains the first set of observations for subgroups 1 to 10, the second record contains the first observations for subgroups 11 to 20, and the third record contains the second observations for subgroups 1 to 10, etc:

```
1.452 1.465 1.538 1.501 1.416 1.581 1.605 1.523 1.563 1.533
1.512 1.499 1.493 1.475 1.542 1.518 1.458 1.534 1.556 1.387
1.472 1.451 1.398 1.494 1.493 1.573 1.533 1.446 1.526 1.542
1.485 1.557 1.542 1.563 1.498 1.519 1.557 1.531 1.519 1.514
1.517 1.521 1.565 1.544 1.431 1.501 1.513 1.487 1.453 1.442
1.562 1.501 1.503 1.516 1.447 1.470 1.550 1.485 1.450 1.496
1.474 1.534 1.555 1.449 1.525 1.460 1.516 1.619 1.513 1.531
1.526 1.488 1.521 1.492 1.461 1.404 1.550 1.533 1.488 1.543
1.520 1.468 1.558 1.418 1.460 1.554 1.495 1.617 1.544 1.588
1.473 1.481 1.518 1.486 1.476 1.508 1.547 1.463 1.538 1.510
```

This file may be read and processed using a FORTRAN-like format. Using this data set, the following commands result in an X-bar chart:

```
FILE wide.data
FORMAT (10(F5.3,X),/,10(F5.3,X))
RDFILE 20 VARIABLES WITH 5 CASES /FORMAT
XCHART_GRF V1 to V20
```

Since the values are separated by blanks, the /RECORDS option on the RDFILE command allows the file to be read in a free-field format. This results in an even easier set of commands to generate the same chart:

```
FILE wide.data
RDFILE 20 VARIABLES WITH 5 CASES /RECORDS = 2
XCHART_GRF V1 to V20
```

If the same data were organized as a single column of values in the file "long.data" as follows:

1.452	Subgroup 1, observation 1
1.472	Subgroup 1, observation 2
1.517	Subgroup 1, observation 3
.	.
.	.
1.496	Subgroup 20, observation 3
1.543	Subgroup 20, observation 4
1.510	Subgroup 20, observation 5

The following commands will use this data set to generate the same chart:

```
FILE long.data
RDFILE 1 VARIABLE WITH 100 CASES
XCHART_GRF V1 /SIZE = 5
```

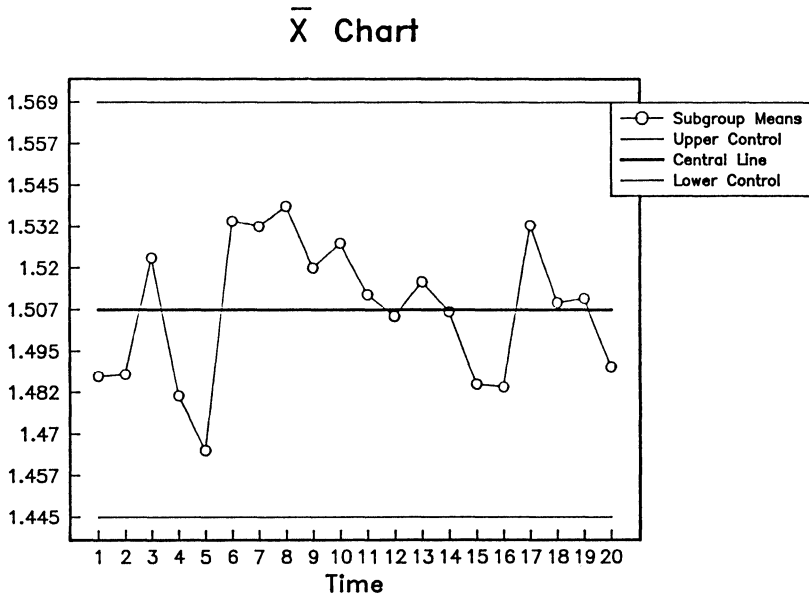
The /SIZE option tells STAT80 to use the first 5 observations as one subgroup, observations 6 through 10 as the second subgroup, etc.

Data sets which are organized by using 'grouping' variables to define the subgroups may also be processed. In the following example, which uses the file "group.data", each of the observations in column one will be in the subgroup defined by column two:

1.452	1
1.465	2
1.538	3
.	.
.	.
1.534	18
1.556	19
1.387	20
1.472	1
1.451	2
1.398	3
.	.
.	.
.	.

The appropriate commands, using this data set, to generate the X-bar chart given in Figure 3 are:

```
FILE group.data  
RDFILE 2 WITH 100  
XCAT_GRP V1 WITH V2
```



Jun. 25, 1988
19:45:22

Figure 3

Data Management

The ability to perform data management is a very important feature of a complete QC program. Data management consists of transforming the data, removing missing values, combining or splitting variables, creating and/or modifying data files, etc.

STAT80 provides extensive transformation abilities with the LET command. This command furnishes over 120 functions ranging from trigonometry calculations and random number generation to a wide variety of

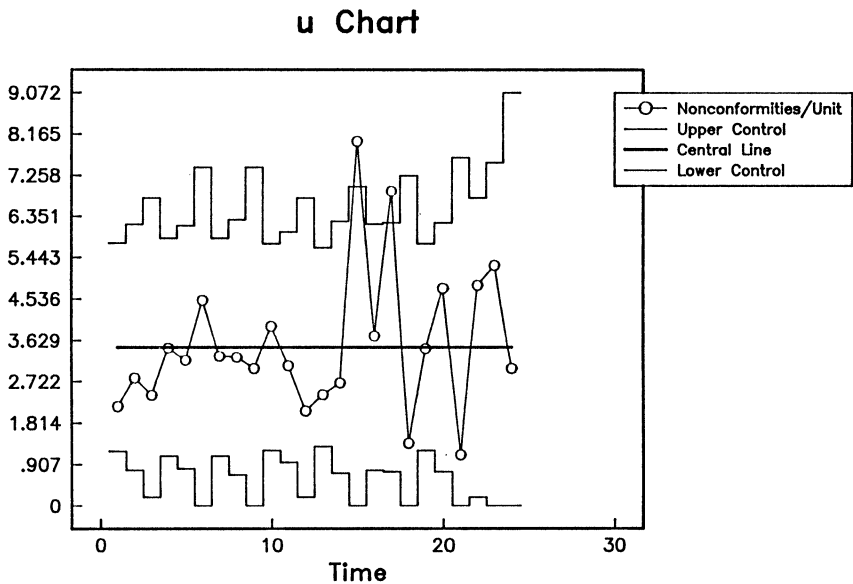
statistical functions.

For example, to change the Celsius data in V1 to Fahrenheit data in V2, the following command may be used:

```
LET V2 = V1 * 9 / 5 + 32
```

STAT80 also provides comprehensive missing data handling, not available in most QC packages.

While many programs can only process subgroups with an equal number of observations, STAT80 furnishes QC charts that will adjust the limits appropriately as shown in Figure 4.



Jun. 25, 1988
19:27:52

Figure 4

Additional data management commands allow the user to combine and split variables.

Data files may be easily combined by STAT80. This may be necessary in order to produce a QC chart using both a data file created a year ago and one created recently. The resulting combined data set may be saved for future use.

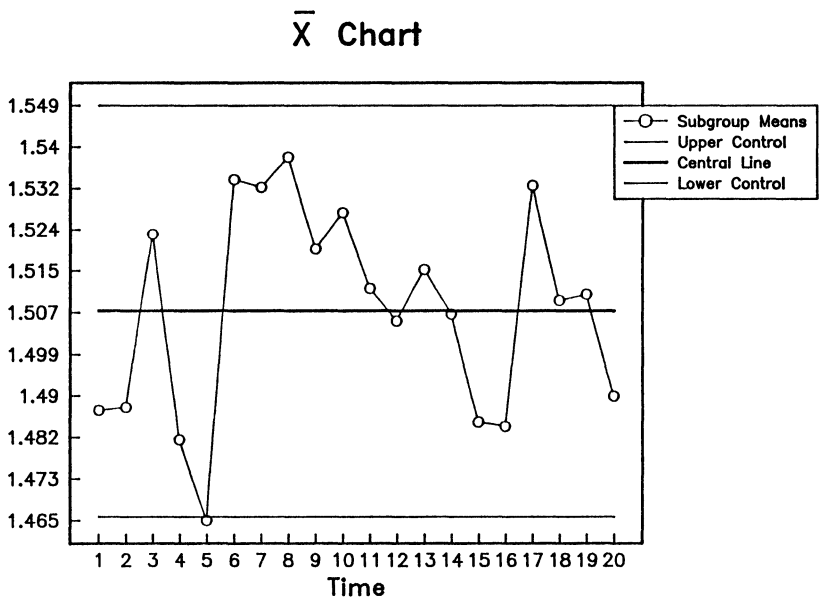
STAT80 also provides the capability to merge (add observations), update (add variables), match-merge (a relational join), and concatenate data sets.

Customizing

Many aspects of the QC chart should be allowed to be adjusted by the user.

For example, the control limits for all STAT80 QC charts are placed, by default, at the industry-standard plus and minus 3-sigma from the central line. Theoretically, this should cover over 99% of the plotted points if the process is in control. In some instances this interval is too wide and should be narrowed. This may be done very easily by using one of the numerous options that the QC procs offer. For example, the command `XCHART_GRF ALL /NSIGMA = 2.0` will place the limits at ± 2 -

sigma on the X-bar chart shown in Figure 5.



Jun. 25, 1988
19:50:33

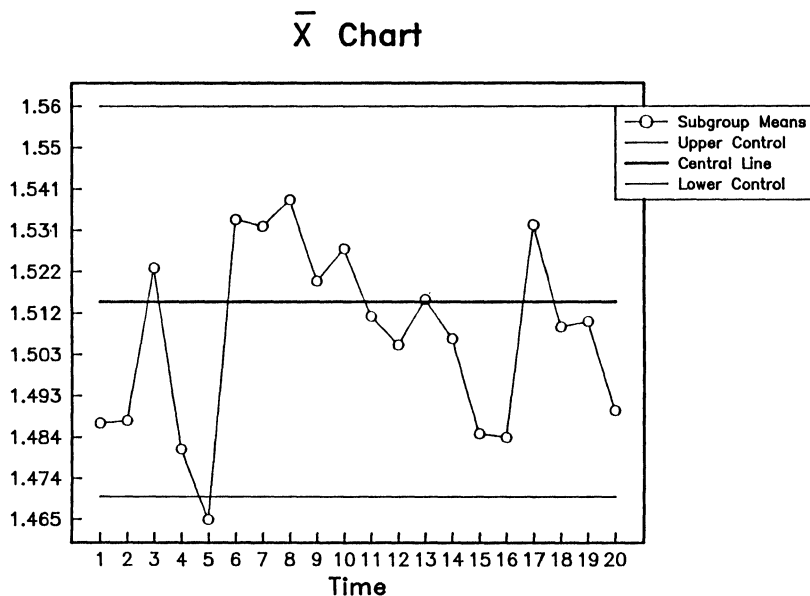
Figure 5

In addition to specifying how far the control limits should be placed from the central line, the user may also dictate where the central line should be located.

If the management or engineers provide standard values at which it is expected that the process can be controlled, the QC charts may be easily customized to reflect these values. For example, if the mean of the process is given as 1.515 and the standard deviation is given as 0.05, then the relevant command is

```
XCHART_GRF ALL /NSIGMA = 2.0 /MSTANDARD = 1.515 /RSTANDARD = 0.05
```

The results of this command are given in Figure 6.



Jun. 25, 1988
20:26:30

Figure 6

This example has illustrated how easily complete control over the limits and central line may be exercised.

Still other options are available that allow the appearance of the chart to be customized. These include the /TITLE, /AXIS, and /NOEXIT options.

The /TITLE option permits the user to specify a title for the chart. The title will be on multiple lines if separate strings are specified; e.g., /TITLE = "Line One" "Line Two". The strings may also include commands called instruction strings. Instruction strings allow the user to turn on and off different fonts such as block, italic, gothic and math (which includes Greek symbols) and also change the character's height and vertical placement.

The /AXIS option allows the user to specify a variable to be used on the

X axis of the plot. The label from this variable will be used as the label for the X axis and may also include instruction strings.

If the previous example is further modified by:

- Creating a variable (V21) for the X axis with a label "May 1986"
- Changing the title to "Customized" "QC Chart"

Then the command, whose output is given in Figure 7, becomes:

```
XCHART GRF V1 to V20 /NSIGMA = 2.0 /MSTANDARD = 1.515 /RSTANDARD = 0.05 ;  
/TITLE = "Customized" "QC Chart" /AXIS = V21
```

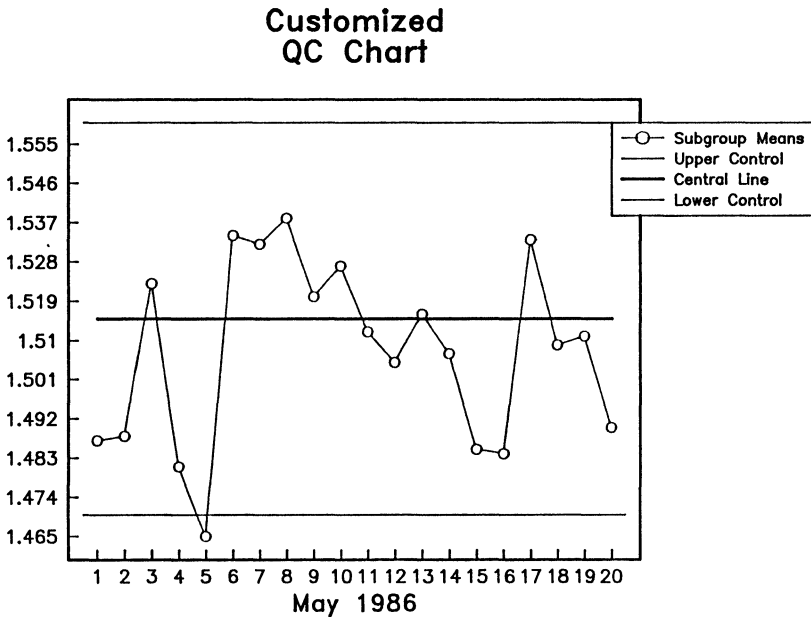


Figure 7

Briefly, the method that the STAT80/GRAFIT interface uses is to have STAT80 build a command file for GRAFIT then execute GRAFIT with this command file. By default, when the graph has been completed, GRAFIT exits and returns control to STAT80. However, there are numerous

capabilities that GRAFIT has that may be more fully utilized through its interactive mode (which requires a graphics terminal). For example, GRAFIT can perform text functions normally found only in word processors. Users may easily format, position, and scale graphic annotation using powerful text generation capabilities such as super/subscripting, underlining, height modification and font selection (GRAFIT has 15 fonts). GRAFIT also allows multiple charts on the same graph.

The interactive mode is often used to modify the chart by adding text, axes names, arrows and changing colors. A popular use of this mode is to mark a point that is out of bounds and give an explanation for its position. This mode is entered by adding the option /NOEXIT to STAT80's command. This option is used to keep GRAFIT from returning control to STAT80 when its command file has finished executing. After the chart has been generated, the GRAFIT banner and prompt "G> " will be displayed indicating that GRAFIT is now in interactive mode.

To enter text onto the chart, follow these steps:

Enter the command:

```
TEXT 1 'Explanation' LOCATE
```

"Explanation" is the text to be displayed on the chart. The command LOCATE results in the locator symbol appearing at the origin. This symbol is then moved, using the arrow keys, to the position on the chart where the text is to be placed.

To put a gold frame around the text, enter the command:

```
TEXT 1 FRAME COLOR GOLD GO
```

This will cause the chart to be redrawn, including the framed text.

Drawing an arrow from the text to a particular point on the chart is done similarly.

Figure 8 illustrates some of the annotation capabilities of GRAFIT.

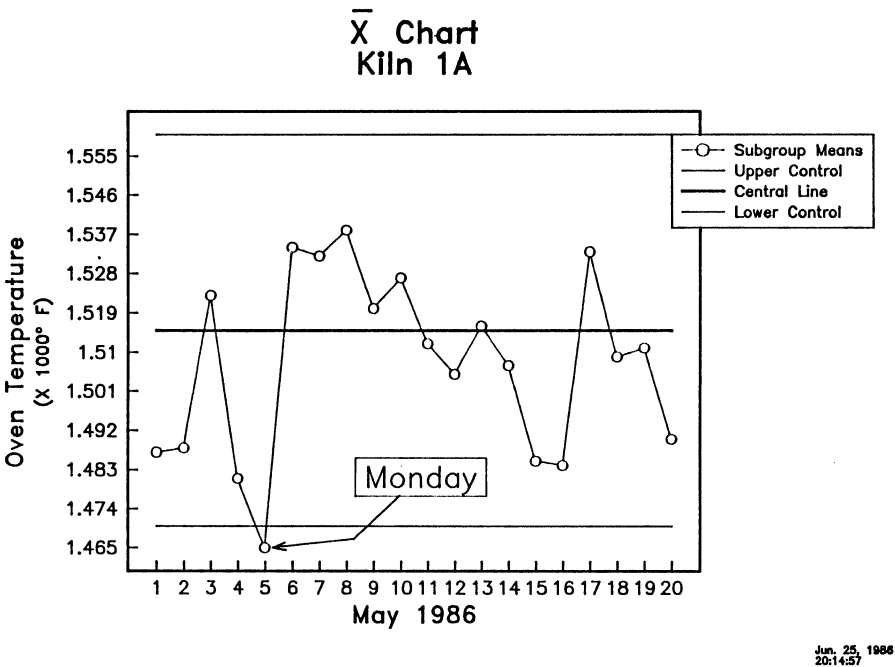


Figure 8

Extensible

In addition to the existing QC charts, users may also write other procedures (procs) using STAT80's internal language. This language closely resembles the "C" programming language yet requires no compiler or linker. Since all source code for the existing QC charts is supplied with STAT80, a user may modify any chart to add options, or use it as a skeleton for creating a different chart. This allows the creation of charts to meet individual needs.

Summary

The STAT80 and GRAFIT combination provides a QC system that is flexible, easy to use, and has extensive data management capabilities. The procedures to generate the QC charts are easily customized and new procedures may be added to the system.

Connection of Black Box Devices
to the
HP1000 A-Series

Wayne R. Asp
Hewlett-Packard Company
2025 W. Larpentour Avenue
St. Paul, Minnesota 55113

INTRODUCTION

In many HP1000 A-Series applications, the need arises to connect some non-HP device, or black box, to the system. Peripheral devices such as printers, tape drives, and terminals, in addition to data acquisition instrumentation and communications equipment, might be classified as a black-box, or generic interface device. These devices usually communicate with the computer system through either IEEE-488 (HP-IB), parallel, or serial (RS-232) interfaces. The IEEE-488 (HP-IB) interface is fairly well defined by the IEEE standard promoting a good level of compatibility between various manufacturer's devices. In contrast, the parallel and serial interfaces lack standardization presenting systems integrators with unique problems when using devices which require these interfaces.

This paper describes the parallel interfacing techniques which are available on the HP-1000 A-Series from both a hardware and software viewpoint using the standard HP supplied I/O Driver.

The 12006A Parallel Interface Card

The 12006A Parallel Interface Card (PIC) is a very flexible interface providing 16 input lines, 16 output lines, 4 device control lines, and 4 device status lines. Two additional control lines are provided to handshake with the device. The PIC also provides external interrupt capability through one status line or one handshake line. Inputs and outputs can be performed using either 8 or 16 bits. Automatic byte packing is provided for 8 bit transfers. The HP data sheet for the PIC is exhibited in Appendix A.

The 40 data, status, and control lines are hardware configurable to TTL(+5V) or +12V operation and software configurable for low-true or high-true sense. This is accomplished by toggling bit 4 (SNS) PIC Control Register 31 (see Appendix B).

Two lines are dedicated to handshaking the data to/from the device. The Device Command (DVCMD) line is controlled by the computer and signals that the computer is initiating a data transfer. The Device Flag, or Service Request (SRQ) line, is controlled by the device and signals the device's completion of the data transfer. When the PIC is armed for device interrupt, the device can interrupt by asserting SRQ.

Four general purpose control lines are provided for computer to device status signaling. These lines are set up on each input or output transfer and are constantly driven out to the device.

The user can force these lines to a given state via a control request to the driver. The driver also has the capability to change line states automatically on read requests or when the card is armed for an interrupt. The user can override the driver and force these lines to any state for any read/write request.

Four status lines are provided for device to computer status signaling. These status lines can be latched on the PIC on each device SRQ signal or be made dynamically available.

Two of the status lines can be used by the device to perform special functions. When configured, status line 0 can be asserted by the device when requesting interrupt service (IRQEN, bit 8, PIC Control Register 31). This bit is normally handled by the driver. An input request can be terminated prematurely by the device by asserting status line 1 if bit 9 (LBYEN) PIC Control Register 31 is set for the input request. To terminate input, the device must assert status line 1 before asserting SRQ.

Device Handshaking

There are 8 data handshake modes available on the 12006A interface to allow the synchronization of data transferred to or from the device. Six control lines can be utilized for the data handshaking. Two of these, Device Command (DVCMD) and Device Flag, or Service Request (SRQ) are required. The other four general purpose control lines can be used to signal the device for specific operations, e.g. input or output transfer.

The DVCMD line is driven by the 12006A card and is used to request the initiation of a data transfer with the device. The SRQ line is driven by the device and is used to signal completion of the DVCMD request for data transfer. DVCMD may be configured for either high-true or low-true operation. U1 Switch 2 configures the DVCMD sense. SRQ does not have a sense, but instead is configured using positive and negative going edges. Bit 12 (DFS) PIC Control Register 31 controls which edge of Device Flag (SRQ) causes the actual SRQ to occur. This is the point at which the data must be valid and/or latched signaling the device's completion of the data transfer.

Full mode and pulsed mode are the two primary modes of handshaking which are configurable. In full mode, DVCMD is asserted to initiate a data transfer and remains asserted until the device has acknowledged transfer completion by asserting SRQ. This is configured by bit 11 (DCL) of PIC Control Register 31 which selects which edge of the SRQ signal clears DVCMD. In pulsed mode, DVCMD is asserted for approximately 227 nanoseconds and then deasserted. The device must then acknowledge the transfer by asserting SRQ. Provided that the device can use pulse mode and meet associated timing requirements, pulsed mode operation can increase data transfer throughput. Full mode or pulsed mode is selected by toggling bit 10 (PLV) in PIC Control Register 31.

Figures 1 through 4 illustrate the eight handshaking modes which are configurable using PLV, DFS, and DCL. The figures shown assume that the 12006A has been configured for high-true logic. Times noted do not necessarily reflect the actual proportional time for operations.

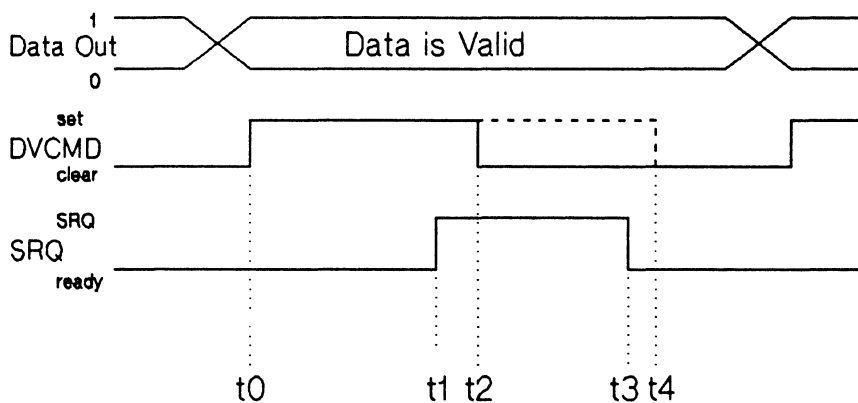


Figure 1: Level Mode Output Handshake

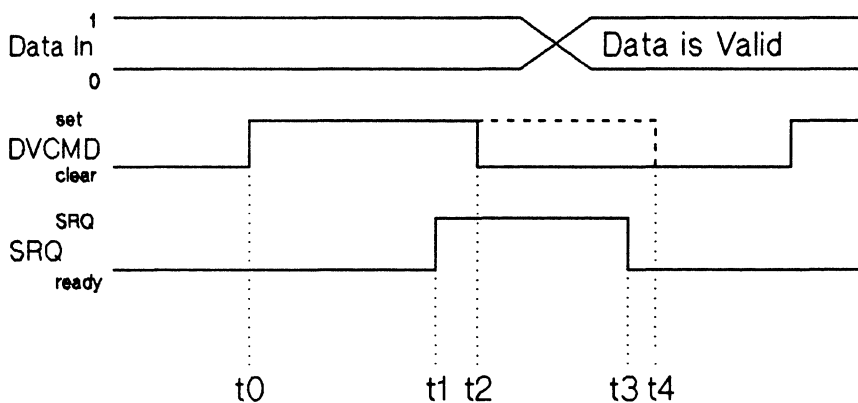


Figure 2: Level Mode Input Handshake

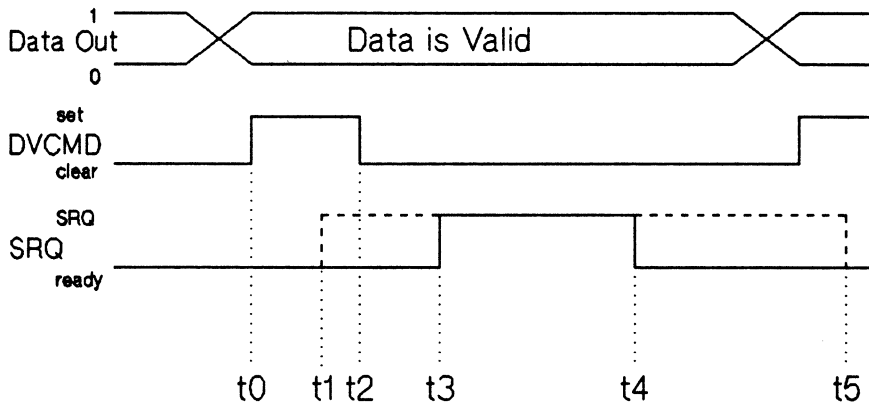


Figure 3: Pulsed Mode Output Handshake

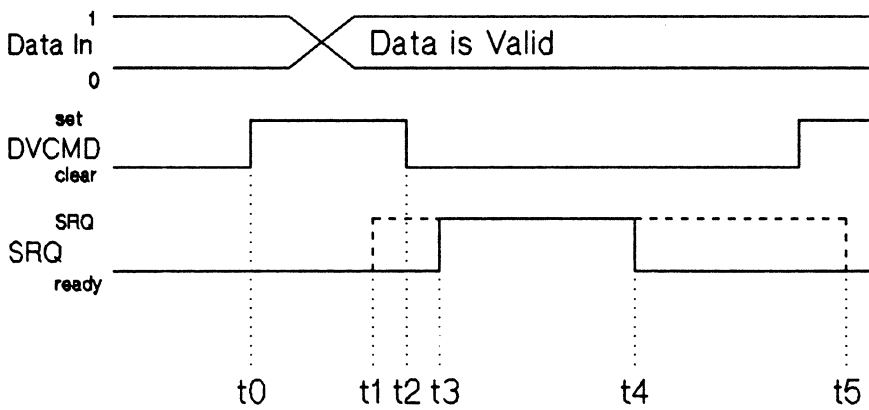


Figure 4: Pulsed Mode Input Handshake

Figure 1 illustrates a level mode output handshake. At time t_0 , the data in the 12006A output register is valid and DVCMD is asserted. At t_1 , the device asserts SRQ. If the device has already latched the output data between t_0 and t_1 , then DCL should be configured for the edge at t_1 which will clear DVCMD at t_2 . In this case, SRQ assertion indicates that the transfer is completed. The card is then ready for the next data transfer. If the device latches the data between t_1 and t_3 , then DCL should be configured for the edge at t_3 which will clear DVCMD at t_4 . In this case, SRQ deassertion indicates that the card has completed latching the data.

Figure 2 illustrates a level mode input handshake. At t_0 , DVCMD is asserted to indicate that the 12006A is ready for data. SRQ is asserted by the device at time t_1 . The device begins to hold data valid on the data input lines between t_1 and t_3 . At t_3 , the device signals data valid by deasserting SRQ. DCL has been configured for this edge. DVCMD is then cleared at t_4 . Alternately, the device can begin to hold valid data on the input lines between t_0 and t_1 (not shown), and DCL is configured for the edge at t_1 . In this case, SRQ occurs at t_1 instead of t_3 and DVCMD is cleared at t_2 .

DFS and DCL also provide some other possibilities. For instance, DVCMD could be cleared at t_2 , but SRQ will not occur until t_3 . Thus, DFS and DCL provide for maximum flexibility for full mode operation. Care must be exercised that DFS and DCL are not configured such that the 12006A is ready for the next transfer before the device has completed the previous one. If this happens, unexpected results or hangs could occur.

Figure 3 illustrates the pulsed mode output handshake. The 12006A latches valid data on the output data lines and asserts DVCMD to initiate the transfer for approximately 227 nanoseconds from t_0 to t_2 . The device begins to latch the data and asserts SRQ somewhere between t_1 and t_3 . The device completes its operation and releases SRQ somewhere between t_4 and t_5 . Note that it is possible for the data to be invalid between t_4 and t_5 if DFS selected the edge from t_1 to t_3 for actual SRQ.

Figure 4 illustrates the pulsed mode input handshake. DVCMD is asserted for about 227 nanoseconds from t_0 to t_2 to indicate that the 12006A is ready for input. The device then outputs valid data on the input data lines. Between t_1 and t_3 , the device asserts SRQ indicating that the data is valid. DFS selects this edge and the request is completed. SRQ is deasserted by the device between t_4 and t_5 . Alternately, the device could output valid data after asserting SRQ (not shown) and signal valid data between t_4 and t_5 by configuring DFS for that edge of SRQ.

As can be seen from figures 3 and 4, pulsed mode is capable of faster data transfers because the next data transfer can be initiated before the SRQ line has been deasserted by the device from the last transfer. This allows the data transfers to be overlapped slightly, giving greater throughput. For maximum throughput, the data must be valid when SRQ is asserted between t_1 and t_3 .

Card Status

The 12006A card status is available from PIC Input Status Register 32. This status register is loaded on each SRQ assertion, but can be made dynamically available by toggling bit 6 (SRM) PIC Control Register 31. The status word includes general purpose control line states, status line states from the device, SRQ and DVCMD states, DMA

state, and diagnostic information. Status Register 32 is retrieved by the driver after each request, and is user accessible, as we will see later.

Sample Device Configuration

Let us assume that we have a device which uses all high-true logic and asserts SRQ when the device is busy and deasserts SRQ when the device has completed its requested action. How should the PIC be configured?

1. U1S2 open indicating high-true DVCMD sense.
2. PIC Control Register 31 Configuration:
 - * CTL0-CTL3 General Purpose Control Lines. Set to 0. These will be controlled from the driver.
 - * SNS Sense Signal Select. Set to 1 for high-true logic.
 - * DRM Data Register Mode. Set to 1 (almost always).
 - * SRM Status Register Mode. Set to 1. Load on SRQ.
 - * PLV Pulsed Level Mode. Set to 0 for level mode.
 - * DCL Device Command Clear. Set to 0. Same edge as DFS.
 - * DFS Device Flag Select. Set to 0. Negative edge SRQ.
 - * All other bits set to 0.

Driver Control of the 12006A Card

Operating system control of the 12006A card is accomplished through the standard interface driver ID.50. Features discussed in this section assume a software revision of DSD5.0 or later. Previous revisions will contain only a subset of the features discussed here.

From a programmers viewpoint, the first thing which must be done is to configure the Card Control Word, Register 31, to match the device. This is stored in DVT parameter word 1 and may be changed at generation time or via a control 40B request. The value in DVT parameter 1 is in the same format as documented in the 12006A hardware manual for Register 31 (see Appendix B). This will be referred to as the default Register 31 value. ID.50 is capable of changing bits in this word before using it. This is mainly done for IRQEN and CTL3-CTL0. DVT parameter word 1 will not be modified except on a control 40B request.

The next step of configuration involves setting DVT parameter word 2 bits 0,1, and 2. Bit 0 specifies a 16 bit or 8 bit interface (see Appendix C). In our previous example, bit 0 would be set to 1 to indicate 8 bit transfer. Bits 1 and 2 work together to determine the AUTO bit value for input and output requests. AUTO on input means that DVCMD will be asserted after the last element has been input. AUTO on output means that no SRQ is necessary to begin the transfer. The normal configuration is no AUTO on input, and AUTO on output, or both bits set to 0. Bits 3,4, and 5 of DVT Parameter word 2 are not used for the 12006A card and should be set to 0.

General Purpose Control Line Signaling

Recall that the general purpose control lines provide four computer to device status signals. These control lines are set up in PIC Control Register 31 and are constantly

driven out to the device once the register is loaded. ID.50 automatically loads Register 31 on each I/O request initiation. Also, Register 31 is loaded when arming for an asynchronous interrupt, if enabled.

For a write request, the default Register 31 value is or'ed with the first optional parameter of the EXEC write request. Thus, the user may specify any bit, including the general purpose control line bits, to be set for that request. If no optional parameter is specified, the default Register 31 value is used.

A read request begins the same as a write request. The default Register 31 value is or'ed with the first optional parameter of the EXEC request. Next, the value in bits 11, 10, 9, and 8 of DVT parameter word 2 are or'ed with the Register 31 value representing CTL3, CTL2, CTL1, and CTL0, respectively. The resulting value is then loaded into Register 31 on the 12006A. The use of these four bits in DVT parameter word 2 allow ID.50 to automatically signal the device for a read versus write request, thus relieving the user from using the optional parameter in the EXEC request. The optional parameter can still be used if desired.

When program scheduling, or asynchronous interrupts, have been enabled, the 12006A is armed for interrupt following any I/O request. Register 31 is loaded just before the card is armed. The Register 31 value loaded is the default value with bits 15, 14, 13, and 12 of DVT parameter word 2 or'ed representing CTL3, CTL2, CTL1, and CTL0 respectively. This allows ID.50 to signal the device when an asynchronous interrupt has been armed on the PIC card.

If the programmer uses the default Register 31 value and configures bits 15-8 of DVT parameter word 2 correctly, the device can examine the control lines and determine what type of request is currently pending; read, write, or interrupt. If more sophisticated signaling is needed, the user can assert the control lines directly through the optional parameter in the EXEC read or write request.

If the user needs to signal the device independent of an I/O request, ID.50 provides the 24B control request. In this request, the lower four bits of PRAM1 are or'ed with the default Register 31 value and the result is loaded on the 12006A. Note that if asynchronous interrupts have been enabled, Register 31 will be immediately reloaded by the interrupt arming upon driver exit from the control 24b request processing. In this instance, the control lines asserted by the control 24B request will remain asserted for only 75 microseconds on an A600+.

An excellent example of control line usage is using the 12006A to communicate with a GPIO device. GPIO requires an input/output line(IO) driven by the computer to signal the device as to the direction of the data transfer. IO is always high-true and asserted only for inputs. In PIC Control Register 31, the interface sense (SNS) would be set for high-true. The IO line from the device would be connected to one of the general purpose control lines, CTL3 for instance. Bit 3 in the default Register 31 value would be set to 0. Bit 11 in DVT parameter word 2 would be set to 1. On a read request, ID.50 would automatically assert (high) IO(CTL3). For a write request, IO(CTL3) would not be asserted (low). This technique allows the device to process both inputs and outputs which are signaled automatically from the driver.

Device Status

After completion of each EXEC request to ID.50, the 12006A Status Register 32 (see Appendix B) is loaded from the card and saved in DVT word 18. This status information can be retrieved in the standard way by making a RMPAR call immediately following the EXEC request.

The status information can be obtained dynamically by executing a control 6B request. The lower 4 bits of the status word reflect the actual states of the device status lines. If the Status Register Mode (SRM) bit is set to 1 in PIC Control Register 31, then the status was loaded into Register 32 on the last assertion of SRQ. Otherwise, the status is "transparently latched" or loaded every 227 nanoseconds.

Program Scheduling

If the device requires asynchronous service, an interrupt program can be scheduled by ID.50 upon receipt of an asynchronous interrupt. Program scheduling is enabled via the control 20B request. When the program is scheduled, the interrupted LU, Register 32 Status, and an optional user parameter word are passed to the program through the run string. In VC+ systems, the program must have been ATACH'ed to the system session prior to the control 20B request. For maximum efficiency, the program should terminate saving resources, although serially reusable and standard termination are also acceptable. Program scheduling is disabled with the control 21B request.

New at revision DSD5.0 is the ability to enable/disable asynchronous interrupts with a control 23B request. Using this request, ID.50 does not "forget" the name of the interrupt program to schedule as with the control 21b request. Thus, the user can enable or disable interrupts as required without supplying the program name every time. This can even be done from the interrupt program itself as will be shown later. When initially setting up scheduling with the control 20B request, a control 23B request is unnecessary as the 20B request will enable asynchronous interrupts automatically.

DVT parameter word 2 bit 6 provides ability to enable/disable the illegal interrupt message. An illegal interrupt occurs whenever an unexpected interrupt is received or a valid interrupt cannot be processed for some reason. This feature should be enabled for most applications.

If DVCMD and SRQ are tied together at the device to show an immediate completion and the DVCMD, SNS, and DFS senses are appropriate, it is possible that the 12006A will become interrupt bound upon receipt of an interrupt, thus hanging the entire system. This is because the 12006A is rearmed for interrupt upon driver exit, which asserts DVCMD, which asserts SRQ (as they are tied together), which generates another interrupt. The easiest solution to this problem is not to tie DVCMD and SRQ together at the device. If this is not possible, there are two options in ID.50.

If the device expects to have DVCMD asserted when the PIC is armed, set DVT parameter word 2 bit 7. This bit tells the driver to clear the first SRQ it receives after DVCMD has been asserted. When the PIC is armed, DVCMD is asserted, SRQ is asserted and then cleared. The next assertion of SRQ will generate the interrupt. If the device does not have DVCMD and SRQ tied directly together and introduces more than

a 6 microsecond delay, the first SRQ will not get cleared and the problem can still occur.

If the device does not expect to have DVCMD asserted on interrupt arming, this can be completely disabled by setting bit 0 in DVT parameter word 3. DVCMD will then not be asserted when the card is armed.

Some devices do not assert the SRQ line when interrupting, but have a separate interrupt line altogether. To accommodate these devices, the 12006A allows the first status line, ST0, to be used as the interrupt request line. To enable this feature in ID.50, bit 1 of DVT parameter word 3 should be set. This enables the driver to automatically set the IRQEN bit in PIC Control Register 31 when the card is armed. The device can then assert either ST0 or SRQ to request an interrupt.

Handling Multiple Device Interrupts

When multiple interrupts occur from a device before the interrupt program can process them all, the driver must take some action to handle the additional interrupts. ID.50 has three user configurable modes to accomplish this.

- * The additional interrupts can simply be ignored. This is the default configuration of ID.50. If the interrupt program is busy and another interrupt occurs, the driver will ignore it and print an illegal interrupt message, if enabled (see previous discussion).

- * The interrupt program has full control over interrupt enable/disable. To use this feature, bit 2 of DVT parameter word 3 must be set. When an interrupt occurs, ID.50 schedules the interrupt program and then disables interrupts as in the control 23B request. The interrupt program then executes. Upon completion of processing, the interrupt program (or some other program or user) must reenables interrupts via a control 23B request. In this case, no interrupts are recognized until the enable control 23B request is executed.

- * The driver has the ability to hold off pending interrupts if the interrupt program is busy. To use this feature, bit 3 of DVT parameter word 3 should be set. When an interrupt occurs, the interrupt program is scheduled and the 12006A is rearmed. If another interrupt occurs before the program has completed processing, the driver disarms the card and holds the interrupt until the program has completed. The interrupt program is then scheduled for the second interrupt and the card is armed once again. The program is checked for completion every 20 milliseconds.

Should the user inadvertently set both bits 2 and 3 in DVT parameter word 3, ID.50 will operate as if only bit 2 was set. For most applications, allowing the driver to hold off pending interrupts will suffice.

Transparent Latched (Dynamic) Reads

In addition to standard read and writes, ID.50 also supports a special 1 word read scheme called transparently latched reads. By setting bit 9 in the CNTWD word of an EXEC read request, the user can transparently read exactly one word or one byte of data from the data input lines. When this bit is set, ID.50 configures the 12006A for transparent latched mode on the data input register. This means that the device does not have to assert SRQ to complete the request as the data is latched every 227 nanoseconds

into the data input register. This is particularly useful for devices which do not support the full or pulsed handshake methods as their data input lines can read dynamically.

A Real Life Example

ABCD Publishing Company has recently purchased a new phototypesetting machine which they want to interface to their typeset production system, an A900. The new machine comes with either a serial or parallel interface option. ABCD decided to purchase the parallel option as they should be able to produce material many times faster than with the serial interface.

The parallel interface on the new machine is specified in the data sheets as follows:

- +12V Device Interface
- 8 data input lines
- 6 data output lines (status)
- 3 command input lines (send commands to machine)
- 1 Computer Ready line (DVCMD)
- 1 Device Ready line (SRQ)
- 1 Interrupt request line

An analysis of the timing diagrams provided show a throughput of 300Kb/sec utilizing full handshake mode. All signals on the interface are low-true, except the Computer Ready Line, which is High-True. When the computer ready line is asserted, the 3 command input lines tell the machine what command is on the data input lines, e.g. command, data, status request, etc. The machine then takes the appropriate action and strobes Device Ready. Status is always dynamically available on the 6 data output lines. The machine requests an interrupt when operator intervention is required to correct some problem.

Configuration of the 12006A includes setting the card for +12V operation and fabricating a cable. The cable will route the 8 data input lines to the lower 8 data output lines of the 12006A. The 6 data output lines will be connected to the lower 6 data input lines for the card. The 3 command input lines will be connected to CTL3, CTL2, and CTL1. The DVCMD and SRQ lines are connected as indicated. The interrupt request line is routed to ST0. Lastly, U1S2 is configured for high-true operation.

In the generation, ID.50 is configured as follows:

```
IFT,%ID*50,SC:35B
DVT,,,LU:54,TO:5000,TX:2,DX:3,DP:1:10040B:102101B:13B,DT:50B
```

Alternately, if ID.50 was already genned in, the control 40B request could have been used:

```
CN,54,40b,10040B,102101B,13B
```

Using some preliminary test programs, ABDC verified that this configuration could communicate properly to the machine.

ABCD then wrote a set of communications programs to handle I/O to the machine. A short excerpt is shown here:

```

subroutine output(command,buffer,length,status)
integer command,buffer,length,status
.
.
.
command = iand( command,17B ) ! mask control bits only
c
c   Output the command and data. The read/write command line
c   (connected to CTL2) is handled automatically.
c
      call exec(2,54,buffer,length,command)
c
c   Now get the status from the status lines transparently.
c
      call exec(1,1000B+54,status,1)
.
.

```

The next step was to set up the interrupt program. Since the driver was configured to hold off pending interrupts, this did not have to be done in the interrupt program. Quite simply, the program needed only to get the current status from the 6 status lines, look up an error message, and display it on the terminal. The program then terminates saving resources. After writing the program(OPERM) ABCD enabled interrupts as follows:

```
CN,54,20B,OPERM,0
```

When enabled for interrupt, the machine expected to see one of the control lines asserted. This line, connected to CTL3, will be automatically handled by the driver. Since the DVCMD line was not expected to be asserted in this instance, this feature was disabled in the DVT parameters.

Summary

The 12006A Parallel Interface card is highly flexible and can be configured to fit most black box type devices utilizing this type of interface. The standard driver, ID.50, can be configured to handle a wide range of functions. These include automatic device signaling on the control lines, asynchronous interrupt with program scheduling, and transparently latched data input, in addition to standard reads and writes. Additional control signaling is available to the user through the optional EXEC read/write parameter. Status information is available to the user either statically or dynamically.

Appendix A: 12006A datasheet

Product Number 12006A

The HP 12006A is a multi-purpose interface for 8 or 16 bit bidirectional data transfers between external devices and HP 1000 A-Series Computers and Systems.

Features

- * TTL (+5V) and +12V interface compatibility
- * Separate 16-bit input and output storage registers
- * Built-in DMA capability offering maximum data rates to 850K words per second on inputs and 730K words per second on outputs
- * Wide choice of programmable operating modes for easy use with instrumentation
- * 8 or 16-bit operation with hardware packing of bytes into or from words
- * Pin compatibility with 12566B/C interface used in other HP 1000 Computers and Systems

Functional Specifications

DATA TRANSFER

Protocol: Transfers either 8 or 16 parallel bits at a time.

Maximum Rate: The following transfer rates can be attained in a quiescent RTE-A environment with the 12006A interface in the highest priority position.

	<u>Input</u>	<u>Output</u>
A600+	850K words/s	730K words/s
A700	790K words/s	650K words/s
A900	740K words/s	500K words/s

Typical CPU to CPU transfer rates will be less than 50% of the output rate.

High Logic Level Choices: TTL (+5V) is standard; removal of six resistor packages converts the interface to +12V level.

Logic Levels and Circuits:

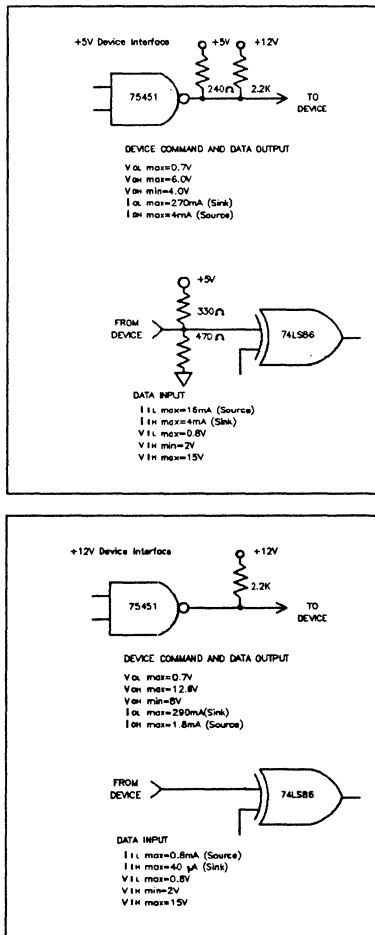


Figure 1. 12006A Logic Levels and Circuits

Byte Packing: For use with 8-bit devices, such as tape readers, tape punches, and some line printers, the interface may be programmed to automatically pack/unpack bytes into/from 16-bit computer words.

Device Command Sense Selection: The interface can be set to respond to either high-true or low-true device command from the interfaced device for card/device synchronization.

Clocked Mode: The parallel interface supports a clocked mode in which data transfers to/from external devices are synchronized by a flag-to-device handshake that is clocked by the external device.

Transparent (asynchronous) Mode: The parallel interface can also be used to send data to or receive data from one or several devices, such as indicators or switches, that do not provide or use any type of clocking signal. Information is output to the destination device(s) exclusively under program control and input information may be read at any time.

CONTROL AND STATUS BIT COMMUNICATION

Control Output: Four control bits may be sent to the interfaced device via an output control word for use as control, command, or address bits. For instance, they can be decoded to address any of 16 device registers or actions, or to address any of 16 devices connected to the same parallel interface.

Status Input: Four status bits may be received from the interfaced device via an input control word.

DIRECT MEMORY ACCESS (DMA) OPERATION

DMA Accessibility: The 12006A can access memory under control of its I/O master processor, regardless of how many other interfaces in the system are also accessing memory via DMA.

Self Configured, Chained DMA Mode: The I/O master processor on the 12006A interface sup-

ports a self configuring mode of operation. In this mode, instead of interrupting the central processor after a block transfer, the I/O processor fetches a new set of control words for the next transfer. This process continues as long as additional sets of control words are available. Chained DMA transfer is particularly useful for storing several sequential scans of measurement channels from an instrumentation subsystem into memory, which can be accomplished without interrupting computations or other processing by the central processor.

CONFIGURATION INFORMATION

Computer and System Compatibility: The 12006A Parallel Interface is compatible with all HP 1000 A-Series Computers and Systems.

Connector Compatibility: The 12006A interface printed circuit cable connector is pin-compatible with the 12566B/C Microcircuit Interface, permitting direct substitution of an HP 1000 A-Series Computer or System with the 12006A interface for an HP 1000 M/E/F-Series with 12566B/C interface.

Software Support: The 12006A interface is supported by RTE-A interface driver ID.50.

Diagnostic Support: A diagnostic and a test hood for the 12006A interface are provided in the 24612A Diagnostic Package.

Installation: Set device command sense switch to appropriate level; set the interface's I/O address on the select code switches; turn off power to the computer and interfaced device; plug the interface into the computer backplane; connect an appropriate cable from the interface to the device; and integrate the interface driver into the operating system if that has not been accomplished previously.

NOTE: The I/O address setting of the interface select code switches is independent of the interface card's position in the computer backplane.

ELECTRICAL SPECIFICATIONS

Direct Current Requirements:

Configured as a +5V Device interface:

+5V at 1.94A

+12V at 179mA

Configured as a +12V Device interface:

+5V at 1.61A

+12V at 175mA

PHYSICAL CHARACTERISTICS

Dimensions: 289mm (11.38 in) long by 172mm (6.75in) wide by 1.6mm (0.063in) board thickness, with 10.2 mm(0.4in) top-of-board parts clearance and 5.1mm (0.2in) beneath-board clearance.

Weight: 370 grams (13oz) with mating connector.

Ordering Information

The 12006A includes:

12006-60003 Parallel Interface Card

5061-3426 48-pin Connector Kit

12006-90001 Reference Manual

Appendix B: 12006A Register Formats

CONTROL REGISTER 31

Register 31 control word is used for both DMA and programmed I/O. Its contents usually depend on the type of device being used.

Two control word examples are as follows:

HP 2895 Paper Tape Punch

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0

HP 2748 Paper Tape Reader

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0

The definitions of the bits in register 31 are as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TSTU	SPARE	DFS	DCL	PLV	LBYEN	IRQEN	TSTL	SRM	DRM	SNS	CTL3	CTL2	CTL1	CTL0	

Bits 0 - 3: CTL0 through CTL3

Four general purpose lines that may be used to address, control, or handshake with various types of peripheral devices. These lines are latched onto the PIC during execution of an OTA 31 (global register enabled) and during DMA self-configuration, and constantly driven out to the peripheral device.

Bit 4: SNS Sense Select Signal

0 = Inverts all 40 status, data, and control signals transferred between the PIC and the device (16 output data lines, 16 input data lines, four control lines to the device, and four status lines from the device). Therefore, this bit, if clear, causes a low true (ground) device interface.

1 = All 40 signals are not inverted, and a high true (positive) device interface exists.

Bit 5: DRM Data Register Mode

1 = Causes data register 30 (input data register, U53 and U83) to be loaded on each DEVICE FLAG signal assertion.

0 = Causes data register 30 to act as a transparent latch so that 16 bits of data are dynamically available.

Bit 6: SRM Status Register Mode

1 = Causes status register 32 (U54, U64) to be loaded on each Device Flag assertion.

0 = Causes status register 32 to act as a transparent latch so that the status is dynamically available.

Bit 7: TSTL Test Lower Bytes

Used by diagnostics to test the operation of the lower bytes of the control and status registers.

Bit 8: IRQEN Interrupt Request Enable

1 = Enables the assertion of ST0 to set the Interrupt Request (IRQ) line, thereby causing Flag 30.

0 = Disables Interrupt Request from being asserted.

Bit 9: LBYEN Last Byte Enable

LBYEN enables early termination of a DMA input transfer.

1 = Enables the assertion of ST1 to cause orderly shutdown of a DMA transfer before word count rollover occurs.

0 = Prevents assertion of LBYEN (last byte), DMA continues until word count rollover occurs.

When this feature is used, ST1 should be asserted prior to the Device Flag accompanying the last word or byte. The assertion of ST1 will not be recognized until the Device Flag Signal is also asserted.

Bit 10: PLV Pulse Level

PLV selects between a pulsed or a level Device Command signal.

- 1 = Produces a Device Command signal, which is a pulse with a duration of one period of SCLK, or approximately 227 nsec.
- 0 = Produces a level mode Device Command.

The Device Command is asserted upon execution of a STC, or automatically during each DMA transfer. The level mode Device Command is deasserted on a selected edge of Device Flag. Also, while DMA is not running, the level mode Device Command signal may be deasserted by clearing the Control flip-flop (executing a CLC 30).

Bit 11: DCL Device Command Clear

DCL selects which edge of Device Flag clears Device Command. This bit only has effect if bit 10 above is clear.

- 1 = Device Command will be cleared on the opposite edge of Device Flag than that which caused the SRQ.
- 0 = Device Command will be cleared on the same edge of Device Flag which caused the assertion of SRQ.

Bit 12: DFS Device Flag Select

DFS selects which edge of Device Flag causes the Service Request (SRQ) signal.

- 1 = Positive-going edge of Device Flag causes SRQ.
- 0 = Negative-going edge of Device Flag causes SRQ.

Bits 13 and 14: Not used (spare).

Bit 15: TSTU Test Upper Bytes

Used by diagnostics to check the operation of the upper bytes of the control and status registers.

INPUT STATUS WORD, REGISTER 32

It is often desirable to interrogate a device as to its status in order to obtain such information as the cause of an interrupt or the state of a control circuit. Sixteen bits of status information are available in a software-accessible status word, which may be fetched using an LIA 32 or LIB 32 with the global register enabled.

Definitions of the bits in status register 32 (U54, U64) are as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TS	SLV	DMA	PACK	CN	CN	CN	TS	TS	FL	DC	ST3	ST2	ST1	ST0	
TS	SLV	DMA	PACK	CN	CN	CN	TS	TS	FL	DC	ST3	ST2	ST1	ST0	

Bits 0 - 3: ST0 - ST3 Status 0 - 3

These four bits provide the PIC with status information from the device to which it is connected. These bits can be

latched into register 32 on each Device Flag assertion or they can be dynamically available, according to the sense of bit 6 of register 31 (see paragraph 3-6).

Bit 4: DCSS Device Command Sense Switch

DCSS indicates the setting of the Device Command Sense Switch.

- 1 = Active high (high true) Device Command (DVCMD) was selected.
- 0 = Active low (low true) Device Command (DVCMD) was selected.

Bit 5: FLAG

FLAG reads the logical sense of the Device Flag line. Initially, FLAG should be zero. This information is most often used for diagnostic purposes.

Bit 6: DFF Device Command Flip-Flop

DFF reads the sense of the Device Command flip-flop. This bit is most often used for diagnostic purposes.

Bit 7: TSTL Test Lower Byte

TSTL is used by diagnostics to check the operation of the lower byte of the control (register 31) and status (register 32) registers.

Bits 8 - 11: CNT0 - CNT3

These four bits check the status of general purpose lines CNT0 through CNT3 of control register 31. CNT0 through CNT3 of control register 31 are used for address, control, or handshake with various types of peripheral devices.

Bit 12: PACK Pack Bytes

PACK is the Q output of the PACK flip-flop.

- 1 = DMA is executing in byte mode, with two bytes packed into each word.
- 0 = Either DMA is executing in word mode, or DMA is not currently running.

Bit 13: DMAON- DMA On

DMAON- is the Q- output of the DMAON flip-flop.

- 0 = DMA is executing.
- 1 = DMA is not executing.

Bit 14: SLV Slave

0 = PIC is in normal mode of operation.

- 1 = PIC is requesting slave mode (access to the Virtual Control Panel code).

Bit 15: TSTU Test Upper Byte

TSTU is used by diagnostics to check the operation of the upper byte of the control (register 31) and status (register 32) registers.

Appendix C: ID.50 DVT Parameters Word Format

Format of DVT Parameter Words:

Word 1: Same as PIC Control Register 31

Word 2: Defined as follows:

- Bit 15: When set to 1, CNTL3 is asserted when the PIC is armed for program scheduling. When set to 0, bit 3 of DVP1 is used for CNTL3 state.
- Bit 14: When set to 1, CNTL2 is asserted when the PIC is armed for program scheduling. When set to 0, bit 2 of DVP1 is used for CNTL2 state.
- Bit 13: When set to 1, CNTL1 is asserted when the PIC is armed for program scheduling. When set to 0, bit 1 of DVP1 is used for CNTL1 state.
- Bit 12: When set to 1, CNTL0 is asserted when the PIC is armed for program scheduling. When set to 0, bit 0 of DVP1 is used for CNTL0 state.
- Bit 11: When set to 1, CNTL3 is asserted when the PIC is set up for a read request. When set to 0, bit 3 of DVP1 is used for CNTL3 state.
- Bit 10: When set to 1, CNTL2 is asserted when the PIC is set up for a read request. When set to 0, bit 2 of DVP1 is used for CNTL2 state.
- Bit 9: When set to 1, CNTL1 is asserted when the PIC is set up for a read request. When set to 0, bit 1 of DVP1 is used for CNTL1 state.
- Bit 8: When set to 1, CNTL0 is asserted when the PIC is set up for a read request. When set to 0, bit 0 of DVP1 is used for CNTL0 state.
- Bit 7: When set to 1, a CLF 30B will be issued to the card after the STC 30B arms the card for asynch interrupt. User having DVCMD connected to FLAG can use this to eliminate continuous interrupts.
- Bit 6: When set to 1, an Illegal Interrupt message will be generated if an interrupt occurs and the program to be scheduled is busy. When set to 0, the interrupt will be ignored.
- Bit 5: DMA Completion interrupts are inhibited on write requests when the bit is set.
- Bit 4: On driver completion exit, do not CLC 30b,C and do not change the card control word if arming for an asynch interrupt when set.
- Bit 3: Ignore optional parameter on read/write requests and place the EXEC code (1/2) into the card control register, or'ed with Driver Parameter 1 when bit is set.
- Bit 2: When set, toggle the DMA AUTO bit in register 21 after processing bit 1 (AUTO on input)
- Bit 1: Set to 1 if the DMA Auto bit is to be asserted on input.
- Bit 0: Set to 1 for 8 bit transfers, set to 0 for 16 bit transfers.

Bit 1 and Bit 2 can be used together to select the AUTO feature for both read and write requests according to the following table.

AUTO States for Bits 1 and 2:

<u>Bit 2</u>	<u>Bit 1</u>	<u>Auto on read</u>	<u>Auto on write</u>
0	0	off	on
0	1	on	on
1	0	on	off
1	1	off	off

Word 3: Defined as follows:

Bits 15-3: Undefined at present.

Bit 3: Driver will hold off pending interrupt if program is not dormant if set. When program goes dormant, it is scheduled and the card is armed for interrupts again. (Note: in this case, timeouts are used. Request timeouts take precedence over pending interrupt timeouts.)

Bit 2: Driver will disable interrupts upon scheduling interrupt program. Program must re-enable interrupts when completed.

Bit 1: When set, enable IRQEN in register 30 (STO interrupts) before arming the card for interrupts, if enabled. Note: if DVTP2 bit 4 is set, this bit will have no effect.

Bit 0: When set, do not assert DVCMD to the device when arming for an asynch. interrupt.

UNDERSTANDING THE NEW SERIAL I/O DRIVERS

Johnny Klonaris
Hewlett-Packard Co.
11000 Wolfe Road
Cupertino, CA 95014

INTRODUCTION

At the 4.1 update for RTE-A and the 5.0 update for RTE-6/VM, a complete set of new serial I/O drivers will be available. These drivers have been completely rewritten. This paper will discuss why the drivers were rewritten, what those changes mean and how users of RTE-A and RTE-6/VM can convert to using these new drivers.

The intent of this paper is not to document specific changes that need to be made as these will be discussed in the appropriate HP documentation. What this paper will discuss is general types of changes that will be likely, what to look for, and strategies for conversion.

I will discuss specifically considerations for

- Black box I/O
- Block mode terminal I/O
- Modem considerations
- Program scheduling

Unless specifically stated, the information in this paper refers to both RTE-6/VM and RTE-A. With that in mind, please remember that this information is not 'official' and is subject to change. Hewlett-Packard will supply documentation when the new drivers and firmware are available. This paper is meant to be an aid to people planning for an upgrade, not to serve as technical documentation.

WHY

There were three major design goals for the Serial I/O driver redesign: improved performance, better reliability, and a consistent interface. The approach taken is discussed below.

Performance

It is fairly well known that the current multiplexers cannot support 9600 baud continuously on all eight ports. In addition, there are other limitations that make the multiplexers and other serial I/O cards unacceptable for communication to some devices, in particular, black boxes. A major consideration of this redesign was to improve

performance, particularly for the eight channel multiplexers (12040 and 12792). We can now support 9600 baud on all eight ports. With the use of bi-directional XON/XOFF we can support 19200 baud on all eight ports. Further, the addition of the control 17b request means that there is much more flexibility in terminating reads which should greatly simplify interfacing to black boxes.

Reliability

This is much harder to qualify. The new drivers and firmware have been designed with structured programming techniques; the External Reference Summary was completed and reviewed before coding began. By release, the testing of the serial I/O drivers will have been the most extensive testing yet done on serial drivers. It is our intent that at release, the new serial drivers will be as 'clean' and 'bug free' as we can make them.

Flexibility was also an issue. The current design is flexible in areas where it doesn't need to be and not where it probably should be. Needed flexibility is good and the current design strives to be flexible as needed. Yet, unneeded flexibility can cause unnecessary complication. The new serial I/O drivers were designed specifically to talk to serial devices. As a consequence, reading and writing to such devices is simpler in nearly every case than with the existing drivers.

Consistency

The differences in interfacing to the ASIC, BACI, and multiplexers, both RTE-A and -6 have been a major stumbling block to applications writers hoping to run their applications on different machines. The differences for block mode applications have been particularly difficult. Another major goal of the redesign has been to create a 'standard' interface; to make it possible to write a program that can do terminal I/O without having to know what system it is running on or what type of interface it is talking to. For the most part, this has been done. In the few places where it is not possible to make everything the same, a standard interface subroutine has been supplied to handle the difference.

WHAT

As one might expect, the above changes have required a change in the interfacing of programs; if an inconsistent interface is made consistent, something has to change. The intent was to minimize the changes needed. In general, standard I/O from Fortran, Pascal, and

Basic programs will require no changes. Simple EXEC calls will also go unchanged. What may require changes are some types of binary reads, HP block mode I/O, and most control requests.

Specifically, this section will discuss the differences between the new drivers and the old in the areas of reads, writes, and control requests.

Read Requests

Read request for the most part are fairly straight-forward. I have broken reads out into three categories: ASCII/Binary reads, block mode reads, and the new special status read.

-ASCII/Binary Reads

The only real difference in the way that reads are done is the handling of the control bits in the EXEC request. There are three bits that I will discuss: the binary, transparent, and echo bits.

The binary bit, if set, causes ALL transfers to terminate on count. This is different for RTE-A users used to using bit 13 in DVP1 which caused binary reads to terminate on receipt of a terminator character. Also, the transparent bit is ignored in binary mode; everything goes into the buffer. If clear, the read becomes an ASCII read, terminating on a terminator character.

The transparent bit now only affects ASCII reads (binary bit clear) as mentioned above. If not set, all is pretty much as it was before with a couple of exceptions: backspaces are now 'destructive' meaning that if 'echo' is turned on a backspace-space-backspace sequence is sent out. This has the effect of 'eating' the end off of the line. Also the terminator character is programable (where the hardware can support it) via the control-17b (more later). If the transparent bit is turned on, all special character processing is suppressed with the exception of the terminator character.

The echo bit is just that - if turned on, characters are echoed back to the terminal, including the destructive backspace mentioned above; if clear, no data are echoed back.

-Block Mode Reads

The block mode process is significantly simplified. In general, the procedure is to strap the terminal to the desired configuration, issue a control-25b request to configure the driver to the terminal, and do the read. The only option is the auto-home bit. The auto-home bit forces an escape-c/escape-H sequence to be sent before reading the terminal. This is a keyboard lock and a home cursor and during a 'user-initiated' transfer prevents the user from 'disturbing' the data

after hitting the enter key. Note that the Auto-home bit is ignored in all modes except block mode.

-Special Status Read

All serial I/O drivers now support a special status read which is an EXEC read of length 32 (words) with all 5 function control bits set (37b). Data are returned to the user buffer specifying all sorts of useful, and previously unavailable data. Such information as the driver name and revision code, the device type, and the returned information from control requests including information like baud rate, protocol, and terminator character.

The existing drivers have been modified to return the fact that the driver is an 'old' one in response to the special status read, allowing an application to determine what sort of interface it is talking to.

This should open up all sorts of possibilities for applications that can talk to any sort of interface.

Write Requests

Write requests have changed very little except that there are now bits in the EXEC control word that effect their operation. These are enhancements and should not effect existing code (unless of course these bits are set in existing code). The bits and a short description of their function follow.

Transparent bit

If clear, an underscore as the last character of a buffer will suppress the CrLf normally following a write. If set, the underscore character will be printed.

Binary bit

If set, ENQ/ACK handshaking, if enabled, will be suppressed for this write.

Force handshake bit

When in 'HP protocol' mode (ENQ/ACK handshaking - see control 34b) this bit, when set, causes an ENQ/ACK handshake to be done before the line is sent. This prevents data from begin lost even if the terminal is turned off or switched to a different line. The main purpose for this bit however is for graphics - some graphics devices do not tolerate the ENQ/ACK handshake well while receiving graphics data.

Control Requests

The majority of the changes have occurred to the control request. Some have been added, some deleted, and many changed: there is a control 17b which adds the capability for user definable terminators on the muxes and to a limited extent on the 12005 ASIC card, the control 52b and 12b requests have been eliminated as there is no longer any need for them - type ahead works in a much friendlier way now, and the control 30b and 33b have changed - this will require most mux users to modify their 'welcome' files. More on this later.

There is a 'Nice' bit in the control 34b request which allows illegal control requests to be ignored rather than flagged as an error. This will allow many applications to run without change. This should be considered a temporary solution at best as it constitutes running with error checking turned off.

The major change for most people, outside of having to modify the welcome file, is the handling of type-ahead or 'FIFO' mode data on the multiplexers. The old way forced a maximum of one record into each of the two 254 character buffers available for each port. Two carriage returns could fill both buffers. This is no longer true. Each port has a 1024 character circular buffer for input. Buffers are filled independent of context - the buffer is not terminated when a carriage return or other terminator character is encountered. This gives a much larger (and much more useful) buffering scheme. Further, a dynamic status request will now return the actual data on the card - not just the currently terminated buffer. This eliminates the need for the control 52b/12b buffer terminate request, and the control 37b and 36b type-ahead buffer terminator and length requests. In many cases, the requests can be removed and the application will run as expected, but this is of course application dependent (or as mentioned above, the 'Nice' bit in the control 34b can be set to allow the application to run without error). The fact that the dynamic status request will return the number of characters currently stored on the card should simplify many applications. The specifics will be discussed in the HP documentation that will accompany the update.

Finally, note that these requests are legal to all of the new serial I/O drivers. It is perfectly acceptable to send a control 30b request to an ASIC or BACI serial card PROVIDED that the parameters passed are legal. Namely, the port number and BRG must be zero and in the case of the ASIC card the baud rate must match the current baud rate of the card or be zero (default). Note that there is an interesting consequence of this. Since the special status read returns the 'returned' parameters of most control requests, it is possible to determine information like baud rate from not only mux cards but also serial cards as well.

The following is a list of control request and a brief synopsis of their function.

Control 6b - Dynamic Status

The length returned in the B-Register is the data currently on the card, as discussed above.

Control 11b - Line Spacing/Page Eject

Designed for serial line printers, this request allows skipping a specified number of lines or a form feed. The form feed can be conditional.

Control 17b - Definable Terminator

This request allows the definition of a user specified terminator character on the mux, or the selection of a ROM based terminator on the A-Series ASIC card (12005).

Control 20b/21b - Program Schedule Enable/Disable

Program scheduling enable/disable is essentially the same except that in RTE-A, the 21b disable request disables all program scheduling, including the control program (typically the 'System' prompt). This eliminates the need for three EXEC calls to disable program scheduling. If either the primary or secondary program is enabled in RTE-A, the control program is enabled.

Control 25b - Read HP Terminal Straps

No real change from the standpoint of use, this request allows block mode transfers to occur - the driver is configured to match the configuration of the terminal.

Control 30b - Set Port ID

There have been two notable changes to the control 30b. First, the ENQ/ACK bit (bit 7) has been moved to the control 34b request. Secondly, there is now an ability to allow speed sensing on the multiplexers. This is done by sending ENQ characters and waiting for ACK characters to return. If the port is connected to an HP terminal using ENQ/ACK, speed sensing is automatic. If the terminal does not respond to an ENQ with an ACK, the port will configure itself when someone starts typing carriage returns.

Note that the value of the parameters is returned from the control 30b and accessible via the special status read. This means that it is possible to retrieve the wiring of the hood as well as the baud rates of the ports. In fact, the baud rate generator (BRG) bit in the request is ignored. The driver determines the actual value from the mux and uses that. The BRG number for a port can then be retrieved with a special status read. There is still a limited number baud rates that can be generated by one BRG. The baud rate

combinations available are a super set of what is available on the existing 'B' and 'C' muxes ('A' muxes had only one baud rate per BRG).

Control 31b - Set Modem Environment

Enables/Disables auto answer and allows connection and disconnection.

Control 32b - Generate Break

Allows programmatic generation of breaks. Note that break is not a character: the data line is held low for approximately 250 milliseconds.

Control 33b - FIFO Buffer Mode Control

This request has changed more than any other. Because of the simplicity of FIFO (or type-ahead) mode, much of what made up the old control 33b, is now gone. What this request does do is:

Enables/disables FIFO mode

Allows incoming characters to be buffered or to cause scheduling

Determines whether or not to keep data upon receipt of a break.

Control 34b - Set Port Protocol

This request used to just turn on or off XON/XOFF protocol. It is now the way to set all protocol for the port. Included is our friend the 'Nice' bit which was discussed above. Additionally, the request specifies if the port is connected to a terminal or hard copy device and what to do with incoming data on timeout. But the main function of this request is the setting of the actual protocol to be used. Of course ENQ/ACK (HP protocol) is available. XON/XOFF is also available on the muxes but now it is bi-directional. Bi-directional XON/XOFF makes it possible for the multiplexer to pace incoming data. This allows very high baud rates (even if throughput may not always keep pace) without ANY data loss.

Control 36b/37b - Type-ahead Length and Terminator.

Gone. As discussed above, these are no longer needed as FIFO mode data are no longer 'terminated' as they come in.

Control 12b/52b - Buffer terminate command.

Likewise. This request (12b for RTE-6/VM, 52b for RTE-A) made data on the card available to the driver. All data on the card are now available to the driver.

HOW

Having all of this wonderful functionality is not of much use if one can't install it. For most, installing the new drivers should be very straight forward; comparable to most system upgrades. For others, it may be a bit more difficult, and for a (hopefully) very few it may in fact not be worth the trouble.

There are basically two considerations here: installation of the new drivers and firmware and conversion of existing transfer files and applications.

Installation

Installing the new drivers should be very easy. It is essentially a matter of modifying the answer file to use the new driver rather than the old one and changing the generation parameters.

The software drivers will consist of the following drivers.

ID100	A-Series 12005 ASIC interface driver
ID101	A-Series 12005 ASIC interface driver with modem support
ID400	A-400 On-Board I/O driver
ID800	A-Series 12040 8-channel multiplexer interface driver
ID801	A-Series 12040 8-channel multiplexer interface driver with modem support
DDC00	A-Series Terminal device driver
DDC01	A-Series Terminal and CTU device driver
DVC00	MEF-Series 'dumb' interface card driver
DVC05	MEF-Series 12966 BACI card driver
DMC05	MEF-Series 12792 8-channel multiplexer driver

Changing the generation parameters is more of a challenge on RTE-A since there really aren't any on RTE-6 (other than specifying the DRT, EQT, and Interrupt table entries and these will change little). Many will recall that 57 words of system table space are required as DVT extension for each port on the existing multiplexer. This number is way down (eleven) with the new driver. The gen parameters are different but are easily determined from existing parameters and documentation.

Another consideration for multiplexer users is the firmware upgrade of the card. In order to use the new drivers it is necessary to upgrade the multiplexer. A low cost kit will be available from HP which will include a ROM to replace the one now on the card, and for A-Series users not on firmware subscription service, the kit will also include updated VCP ROM's.

Once a multiplexer has been upgraded with the new ROM it becomes a 'D'

model and WILL NO LONGER WORK WITH THE EXISTING MUX DRIVERS. So, for this reason, and the fact that an HP Customer Engineer must install the ROM's, it is important that the user have a system generated and ready to go before the new ROM's are installed. Note that the new VCP ROM's will work with either the new or old drivers and that a system can have both kinds of drivers and mux cards installed in the same system.

Details of ordering and installation will be available from HP Support Offices. Most supported HP-1000 customers with muxes should have been contacted by the time of INTEREX or shortly thereafter.

Conversion

Conversion of a system to the new serial I/O drivers will vary in complexity from nearly trivial to very involved. Users whose applications do only standard reads and writes from higher level languages will most probably only have to change their welcome files. Users twiddling bits in the DVT or EQT in hundreds of custom applications may find themselves with a significant task ahead of them. Here we are going to take a look at what kinds of considerations are involved for different types applications.

-Configuration Files

The control requests to configure the multiplexers are most probably going to have to change. For most installations, this will require clearing bit seven (ENQ/ACK) in the control 30b, rebuilding a new control 33b parameter, as well as one for the control 34b. Also, many may want to take advantage of the new speed sensing capabilities by changing the baud rate field in the control 30b.

-Standard I/O

This includes simple reads and writes from applications written in FTN7X, PASCAL, or BASIC or simple EXEC calls that only make use of the echo and binary bits. The vast majority, if not all, of these programs will not have to be changed at all. This was a design goal of the project.

-Black Box I/O

Talking to the proverbial black box is where we've seen the most use of the old 'FIFO' mode - namely that involved example in the manual where the mux is configured first for binary mode, then for a length of 254 bytes and is then alternately 'terminated' and read from. I know, I've written a couple of these myself. The good news is that none of this is necessary anymore. The bad news is that none of this is necessary anymore. It means taking code out of a working program;

a scary proposition at best. However, since most of these applications look something like this....

```
set up for binary type-ahead (control 37b)
set read length to 254 (control 36b)
terminate buffer (control 52b/12b)
loop-start
  dynamic status, any data? (control 6b)
  if data then
    read the data
    terminate a buffer (control 52b/12b)
    - take some action -
  else
    - take some other action -
  endif
  time-delay if appropriate
loop-end
```

Most will end up looking something like this...

```
loop-start
  dynamic status, any data? (control 6b)
  if data then
    read the data
    - take some action -
  else
    - take some other action -
  endif
  time-delay if appropriate
loop-end
```

This is typical of 'full duplex' applications where it is important not to issue a read until data are available because it may be necessary to send data out. We do expect that most black box applications will only require that code be taken out. Note that the first example would work fine with the new drivers if the 'Nice' bit were set in the control 34b; all that was taken out were the control requests that are no longer supported.

-Block Mode Considerations

I know that there are many applications out there that required many long hours to perfect to get around some quirk that came up in some corner case: getting the cursor to home, reading multiple fields in block-line mode, or trying to get one application to read from more than one type of interface. I wish I could tell you that all of your applications will run without change on all interfaces but this is not so. However, we expect that most will. If your application uses a control 25b to configure the driver, chances are very good that it will require little or no modification. It is now possible to write a

program that will do block mode transfers in a straight forward way and run on serial cards or muxes on RTE-A or RTE-6/VM.

There is one important difference. All existing HP serial drivers strip out the unit separator (US) characters out of the buffer except the BACI card driver DVA05 on RTE-6/VM. Since newer terminals now support a 'modified fields only' mode in which unchanged fields are not sent, this is no longer a very useful feature. All of the new serial I/O drivers leave all US and RS characters in the user buffer. The routine HPCrt_StripChar will be supplied to duplicate this function of DVA05. (HPCrt_StripChar can also be used to remove RS characters or any other character for that matter.)

Our expectation is that nearly all block mode applications will continue to run without modification except as noted above for DVA05 based applications.

-Modem Considerations

Modem use has been simplified. The control 30b request still has a bit that indicates a port is a modem port for the 3721⁴ modem card cage. The only other control request having to do with modems is the control 31b. Two bits are used: one for answering and one for connection. If enabled for answering, program HPMDM is scheduled when a ring is detected. HPMDM is the standard name for the program to be scheduled when ever a modem 'event' occurs - ringing or hanging up. By moving most of the modem functions to the HPMDM program, many of the earlier complications should go away.

The HPMDM program will be supplied with source.

-Program Scheduling

As was mentioned above, program scheduling on RTE-A is slightly different. Specifying the scheduled program at gen time is now a matter of encoding a word that contains indices of the programs in a table. The most commonly used programs are already in the table (PROMT, CI, CM, FMGR, and COMND). This means that the six words that used to take up space as driver parameters are now reduced to one word.

Specifying programs to be scheduled on interrupt from CI is the same as before. To make things easier, there are two HP supplied routines named HPCrtSchedProg and HPCrtSchedProg_S the make specifying these programs even easier. The '_S' version takes Fortran CHARACTER type variables as input.

-Documentation

I bring this up here because it is critical to any conversion effort. There is more documentation involved with the serial I/O project than

any I've seen and it is good quality documentation. Two of the three engineers working on the project are ex-HP SE's so they are very familiar with the frustrations we've all had in the past. The documentation should be very helpful to anyone who reads it.

-HPCrt.Lib Routines

Along with the drivers and documentation there is a supplied library of routines that do many of the functions that are needed when doing serial I/O. This partial list is presented just to give an idea of the functionality supplied.

HpCrt_SSRCDriver	- Does driver support Special Status Read
HpCrt_SchedProg	- Set up interrupt program
HpCrt_StripChar	- Remove specified character from a buffer
HpCrt_GetField_S	- Get a character field
HpCrt_GetField_I	- Get an integer field
HpCrt_Page_Mode	- Configure for page/block mode
HpCrt_Line_Mode	- Configure for line/block mode
HpCrt_Char_Mode	- Configure for character mode
HpCrt_CommandStack	- A command stack routine
HpCrt_SendString	- Write a Fortran character variable
HpCrt_GetString	- Read a Fortran character variable
and many more...	

FINAL NOTE

A lot of time and effort have gone into this project, mostly by people that have had experience with the existing serial I/O drivers, to make this system the best it can be. We hope that as many users as possible will be able to change over to these new drivers soon. Yet, at the same time we realize that not everyone will be able to convert. Rest assured that the old drivers are still there and will be there and they are still supported. But remember also that there will be no enhancements to the old drivers. Also, only critical bugs will be fixed in the future as the new drivers fix the bugs we know about. Also note that some 'bugs' can't be fixed because they are being used by others as 'features'. We won't change the functionality of the existing drivers for this reason - you can consider them as 'frozen' code from the point of current functionality.

We hope that these drivers and firmware will solve most if not all of the problems we know many of you have been living with and should greatly ease the development of new applications. Good luck.

Understanding the New Serial I/O Drivers

by: Klonaris, Johnny

We regret that this paper
was not received for
inclusion in these proceedings.

A SET OF IMAGE/1000 DATABASE TOOLS FOR SCREEN ACCESS, DATA MANIPULATION AND STATISTICS

Edward J. Kulis and Nthantu Le
Collagen Corporation
2500 Faber Place
Palo Alto, CA 94303

At Collagen Corporation, we have developed three general purpose software tools for HP1000 IMAGE database programmers and sophisticated users. The first tool, GSA (Generic Screen Access), allows the implementation of an interactive screen-oriented IMAGE/1000 database without any programming. The second tool, DBMAP, provides extensive capabilities for data movement and reformatting within and between IMAGE/1000 datasets and databases. The third tool, STATS, provides basic statistics, and percentages for grouped numeric or character field ranges in records selected by HP's QUERY. In this paper we'll begin with a description of our hardware and provide an historical summary of the evolution of our database capabilities. Then, we'll describe in detail our database requirements, the tools that we purchased and the software that we developed to fulfill them. We'll also comment on some limitations of the tools and software, and we'll illustrate some tricks to enhance their power. At the present time, we have a versatile mature system of tools that permits us to accomplish most database tasks without any FORTRAN programming.

HISTORICAL SUMMARY

Clinical Database Requirements

Collagen Corporation manufactures implantable materials for the repair and augmentation of human tissues. We test the safety and efficacy of new products in human clinical trials. We use the IMAGE/1000 database on an HP1000 A600 to track the progress of our clinical trials and to record efficacy data for the material under test. The database task is complex and a complete definition of data items, coding, selection, sorting, and reporting requirements is not possible before database implementation. In addition, our clinical report forms are compact documents that contain many pieces of information on each page. Figure 2 illustrates the quantity of information contained on a typical clinical report form. The compact forms reduce the paper flow required during a clinical trial.

The sophistication of our tools to enter, retrieve, and process data in response to the complex and dynamic requirements of clinical studies evolved through the 5 phases listed below.

Evolution of Database Capabilities

Phase 1: We learned to use the IMAGE/1000 utilities, DBDS, DBBLD, QUERY, DBULD, DBLOD. IMAGE on the HP1000 was a great improvement in cost and performance over the consultants and the time sharing service that we had used but we also discovered that IMAGE had some limitations. We learned some tricks to enhance our ability to manipulate and report data, but we realized that we needed more capability.

Phase 2: To extend out database capabilities, we purchased the SOLUTION software system from Polaris Systems, Inc. (Manasquan, NJ). Before the purchase, we had also considered QUESTOR and QREPORT from Combs and LaRobardiere, Inc. (Ridgewood, NJ) and TERMINAL MANAGER and ASK/1000 from Corporate Computer Systems, Inc. (Holmdel, NJ) but after considerable analysis we decided that SOLUTION most closely matched our requirements. SOLUTION consists of three packages: 1) INSIGHT, a screen-oriented report generator for the general database user, 2) VIEW, a programmer tool that simplifies the use and documentation of terminal screens, and 3) DIMENSION, a programmer tool that simplifies and documents the programmatic interface between VIEW screens and the IMAGE/1000 database. Again we found that while our capabilities had improved there were still limitations that we wished to overcome.

Phase 3: To simplify database access and to allow the implementation of a screen-oriented database without any programming, we developed GSA. Currently in daily operation, GSA provides a general purpose interface to our IMAGE/1000 databases and it allows a non-programmer to design completely and to implement a database and its screens using the tools, DBDS, VIEW, DIMENSION, and GSA.

Phase 4: To enhance the power of the report generators INSIGHT and QUERY we designed and developed the FORTRAN 77 program DBMAP. Both report generators have strong and weak points. We wished to supplement the weak points and we did not want to re-invent capabilities that we already had. DBMAP provides extensive capabilities to move and recombine data among datasets. Thus, INSIGHT and QUERY can select and sort the recombined data and provide formatted reports that employ data from multiple datasets.

Phase 5: To provide statistical summary reports from our databases we developed the FORTRAN77 program STATS. STATS provides group counts, averages, standard deviations, standard errors, minimums, maximums for 20 variables from records in a QUERY select file. STATS also provides row, column, depth, and total percentages for three variables and 20 ranges. STATS can also exclude missing values. There are statistics packages available for the HP1000 (e.g. STAT80 from STATWARE, Salt Lake City, UT) but we

wished to develop one that used data directly from an IMAGE/1000 dataset.

DEVELOPMENT OF OUR CLINICAL STUDIES DATABASE CAPABILITIES

HARDWARE

Figure 1 contains a list of our computer and communications hardware. An HP1000 A600 running the RTE-A operating system has served our database needs since December of 1983. A 7914TD provides 132 Mbytes of disk storage and a 7970 9-track tape drive; a 7946A provides an additional 55 Mbytes of storage and a cartridge tape drive. We have 2 Mbytes of memory, 2 12005B asynch cards and 2 12040B 8 channel MUX cards. Our peripherals include 6 2392A terminals, 2 printers (a 2631B and a 2934A), 9 HP9816 desktop computers, 3 IBM PC clones, and 1 IBM XT. The HP9816's function mostly as laboratory workstations but they also emulate HP2622 terminals to provide the laboratories with access to centralized IMAGE/1000 databases. The IBM clones function as personal workstations and the IBM XT runs BMDP Statistical Software from BMDP in Los Angeles, CA for our advanced statistical needs. The IBM machines also emulate HP2622's and freely exchange data with the HP1000. All terminals, printers, and computers, distributed throughout 2 buildings, communicate with the HP1000 via an RS232 Distributed Data Switch from Metapath, Inc. (Foster City, CA).

CLINICAL STUDIES DATABASE REQUIREMENTS

A Hypothetical Clinical Study

To illustrate the requirements of a clinical studies database, we'll describe the conduct of a hypothetical clinical study. This study, its clinical report forms, and its database are much simpler than real ones but they will serve to illustrate the key database concepts and software tools that we use.

Figure 3 contains a schematic representation of the course of a hypothetical clinical study. This study examines the ability of a product to correct a facial defect such as a wrinkle or an acne pit. The doctor treats the patient with 1 of 2 products at 1 or more sites. Correction level is the parameter of efficacy and both the patient and the doctor estimate the level of correction for each site on a 1 worst to 5 best scale at the time of 1 to 3 treatments. The patient and the doctor estimate correction remaining using the same scale at 5 evaluation visits scheduled over the 2 years following the last treatment. The 5 evaluation visits occur at 7 weeks, 3 months, 6 months, 1 year, and 2 years. The doctor and his staff complete 4 types of clinical report forms during the study. The first form, completed once, records the enrollment of the doctor. A

second form, completed once, records the patient's informed consent and medical history. A third form, completed at each treatment, records the materials used, the sites treated and the estimates of correction. Similarly, a fourth form, records the estimates of correction remaining at each of the evaluation visits.

Figure 4 contains a list of database requirements during the conduct of a clinical study. The tasks are listed in the approximate order that we were able to implement them on our HP1000. With the implementation of each new computer capability we increased our productivity because we reduced the amount of hand collation and calculation required during a particular task.

Image/1000 Database Schemas and Clinical Studies Data

The ability to retrieve information from any computer database depends on the structures available in the database, the user's design of the database, and the coding conventions employed during data entry. IMAGE/1000 provides a simple network structure which is suited for use in clinical studies. A single patient returns to the doctor for multiple treatment and evaluation visits. IMAGE/1000 can represent the patient as a data entry at the head of a chain in a master dataset. This master entry can also record information that is related to the whole patient such as informed consent and medical history. Data entries along each patient chain in a detail dataset can then record information related to each visit. In addition, appropriately chosen visit codes can sort each patient chain in time. Applications programs can take advantage of these sorted chains because the order of record access is well defined.

While IMAGE/1000 provided a logical structure to represent clinical studies data, it did not provide the utilities to take full advantage of its structure! As a result, as we acquired more database tools, the design of our database schemas changed to take advantage of the properties of the IMAGE/1000 structure and our new tools. Figure 5 summarizes the database design considerations that led us to use 4 different types of database structures during the evolution of our database capabilities. Figure 6 lists specifically the elements included in each type of database schema design and Figures 7 through 10 each contain an annotated part of an IMAGE/1000 schema that illustrates the key concepts that we employed in our clinical databases at each stage of development.

PHASE 1: IMAGE/1000 and HP UTILITIES

IMAGE/1000 is in one sense a complete system: QUERY can add, replace, delete, and report data immediately after the DBDS utility builds the database from a schema designed by

the user. This immediate ability to access and retrieve data can be very useful at times when the need for a computer database is urgent. In this section, we'll describe the benefits gained and the limitations that we discovered as we began to fulfill our clinical database requirements with IMAGE/1000.

The Database Schema for the Hypothetical Clinical Study

Figure 7 contains an annotated part of an IMAGE/1000 schema that illustrates the key concepts that we employ in our clinical databases. The schema in Figure 7 was optimal for use by QUERY. The 4 datasets, QDMD, QDHIST, QDTREA, and QDEVAL, correspond to the 4 hypothetical clinical report forms for the MD information, and each patient's visits for history, treatment, and evaluation. As each form arrives its data is added to the database via QUERY's UPDATE ADD. Information from different forms resides in different datasets because each form arrives at a different time and each contains different information about an MD, patient, or a patient's sites.

The data items MDNO and PATNO are key items in master datasets. Since a master dataset can have only one value of a key item, each MDNO and PATNO is unique. These key items define data chains in the detail datasets QDTREA and QDEVAL and thus prevent the addition of an MDNO or PATNO that does not exist in the master. However, the entry of a mis-matched but valid MDNO for a given PATNO is possible. In the master dataset QDHIST, CUSTNO contains the MDNO for each patient. MDNO cannot be used in QDHIST because it is a key item in the master dataset QDMD. CUSTNO allows QUERY to sort patient histories by MD number.

In both QDTREA and QDEVAL, the item PTRECL provides a way to identify the patient's first site at the particular visit. Since each data entry contains information about a particular site, there is no way to determine the number of patients that have been treated or evaluated without the PTRECL item. The data entry operator enters a Y into PTRECL for the patient's first site at a particular visit. PTRECL is left blank for the records of the remaining sites at that visit.

In QDEVAL, the item DAYCNT is the number of days between the last treatment and the current evaluation. When QUERY alone is used, the data entry operator computes this value manually from the date of last treatment and enters it.

Benefits

With IMAGE/1000, we could fulfill the first 3 requirements in Figure 4 for a clinical study database. With QUERY we could add, modify, retrieve, sort, and report data from each clinical report form. We could add items to the data-

base during the course of the study by unloading the data with the utility DBULD, changing the schema, and re-loading the data with DBLOD.

Limitations

Data entry via QUERY is line oriented, and user prompting is limited to the 6 character item name. (The cryptic prompt lengthens the training time for a data entry operator.) QUERY searches are limited to a single dataset and the searches cannot compare one item to another. At the time, find statements could not contain wild-card characters and reports can contain only 10 output lines per record. Arithmetic manipulation is limited to counts, totals, and averages on a single item. The ability to add items to a dataset is also limited. DBLOD can only add items to the end of a dataset and this may be inconvenient for the operator since during an UPDATE ADD, QUERY presents items in the order that occur they in the schema and the new order may not correspond to the order of items on the clinical report form.

Techniques and Work-Arounds

Although DBLOD cannot add items to the middle of a dataset, a combination of other HP utilities can. The technique involves the use of QUERY, EDIT/1000, and another IMAGE/1000 utility DBBLD. DBBLD can add data from a column formatted ASCII type 3 file to a dataset. QUERY can produce a report file with data from the old dataset aligned in the proper columns for the enlarged dataset. EDIT/1000 can remove spurious characters (such as the formfeed characters in column 1) from the file and provide a way to insert DBBLD instructions into the file. Then, DBBLD can add the data in the file to the new enlarged dataset.

PHASE 2: THE INSIGHT REPORT GENERATOR FROM POLARIS

Benefits

The first part of the SOLUTION package that we implemented was INSIGHT. We wished to perform wild-card and comparative searches and multiple item arithmetic manipulations in reports longer than 10 lines from multiple datasets. INSIGHT made this possible. We were able to select data with wild-card find statements from as many as 6 datasets and produce reports with as many as 60 lines per page. INSIGHT'S threading feature enables access to as many as 10 additional datasets for each dataset referenced in a find statement. We could select data based on a comparison of items within a dataset. For example, we could find all records in QDTREA (Fig. 7) in which the patient's estimate of correction did not match the MD's. INSIGHT's register arithmetic manipulation enabled us to produce reports that calculated statistical standard deviations in addition to

counts, totals, and averages. INSIGHT provided us with many new capabilities and it has proved a valuable tool. In addition, its screen-oriented user interface simplifies data access by the occasional user.

Limitations

When we attempted to produce full patient reports with INSIGHT, we discovered that though we could produce the reports sorted by patient, we could not produce them sorted by patient within MD. The problem occurred because INSIGHT connects or links records in different datasets by a single data item and that item must also be the highest sort item. The proper item for full patient reports sorted by MD, a concatenation of MDNO and PATNO, was not present in our datasets! Figure 11 illustrates this problem. If the connector item was PATNO, INSIGHT could produce a proper report containing the data from each patient's history form followed by the data from the patient's treatment and evaluation forms but the MDNO associated with each patient was different from one patient to the next. If the connector item was MDNO then the data from all patient histories for a particular MD would group together, followed by the data from all treatment forms for all patients of each MD and that was followed by the data from the all evaluations of that MD's patients. Thus, although the report was sorted by MDNO, data was grouped by form and not by patient. Manual collation of the full patient reports by MD was tedious and time consuming.

We also discovered another limitation of multiple dataset finds and reports. The link item between datasets is the highest sort item. Changes in value of sort items define group breaks where group operations are performed, and higher group breaks force lower group breaks. So after selection of records from 2 datasets all group breaks occur when the link item changes value. Since the link item must be specific to match records in different datasets, the power to perform grouped arithmetic operations is limited. In essence, only final total operations are still useful. A description of a particular reporting task will illustrate this limitation.

Refer to the schema in Figure 7, during the following discussion. Suppose that we wish to average the volumes used for each treatment in the QDTREA dataset and that we want the averages for females only. The dataset QDHIST contains the sex of the patient. INSIGHT can find all SEX is F in QDHIST and use PATNO to link exclusively the records in QDTREA that contain matching PATNO's, but since PATNO is the highest sort item, INSIGHT can only group visits under patients. It cannot provide the grouping of patients under visits that is required for the calculation of average volumes for females at each visit. INSIGHT can provide the averages in the final total lines in separate reports, one

for each visit. However, multiple reports are less convenient than a single grouped report.

Techniques and Work-Arounds

To enable INSIGHT to produce full patient reports sorted by MDNO, we added an item named IDNO item to all patient datasets using the techniques described above. IDNO is the concatenation of MDNO and PATNO. We retained both the items MDNO and PATNO because we still wished to link and sort by them individually. Figure 8 shows an optimal schema for use by QUERY and INSIGHT and Figure 12 shows a short FORTRAN 77 program using IMAGE/1000 calls to produce an item in a dataset from the concatenation of 2 items in the same dataset. With the addition of IDNO, INSIGHT was able to produce the full patient reports sorted in the manner that we desired. We did not solve the problem of fully flexible multiple dataset reporting until we developed DBMAP, which we'll describe below.

PHASE 3: GSA (GENERIC SCREEN ACCESS)

Benefits

To obtain the benefits of screen-oriented database access and to overcome the limitations inherent in QUERY's line-oriented access we developed the FORTRAN 77 program GSA.

Screen-oriented database access enhances the use of clinical studies databases in a number of ways. Data entry operators require less time to become familiar with a new database because screens that look similar to the clinical report forms help orient the operator. Actual data entry is easier because the operator can check the screen and correct errors before posting the screen to the database. Data coding is also much easier because the screen can display the coding conventions. Occasional database users especially find screen access far easier than line access.

GSA makes extensive use of Polaris's programmer tools, VIEW and DIMENSION and these tools greatly simplified the programming and documentation needed for us to develop a general purpose database access tool. VIEW provides an interactive way to design and store terminal screens for later use. GSA or any program can then access the screens through VIEW subroutine calls. VIEW eliminates the need for a programmer to deal with all the escape sequences and control codes necessary to format a terminal screen and VIEW stores the screens by name in a forms file and thus removes the need to store screen specifications within a program. Instead, the program can reference the screen by name. DIMENSION provides an interactive way to map the windows on a VIEW screen to an IMAGE/1000 database. A program can then invoke a screen and its relation to a data -

base with a single transaction number. Days or weeks of tedious programming work are eliminated and changes to a screen or a database require only changes in the VIEW and DIMENSION configurations instead of program modifications and re-compilations.

Although VIEW and DIMENSION provide powerful tools to move data between terminal screens and IMAGE/1000 databases, a shell program must be developed to orchestrate the use of the tools and to co-ordinate the database access. We did not wish to compile, document, and maintain different programs for each database, so we designed and developed GSA, a generic program that provides access for any database that we design.

GSA removes the need to write and maintain separate shell programs for each database. GSA reduces database implementation to 4 steps: 1) Design an IMAGE schema and create a database, 2) With VIEW, design one top menu screen for the database and one or more data entry screens for each dataset, 3) Map the dataset screens to the database with DIMENSION, 4) Map menu numbers to DIMENSION transaction numbers with GSA. Once an IMAGE/1000 database and VIEW screens are designed, DIMENSION and GSA mapping require less than 15 minutes per dataset.

As depicted in Figure 13, GSA provides a standardized type of screen access for our IMAGE/1000 databases. The top menu screen, contains contains an annotated list of menu numbers and each number allows selection of a particular screen for each dataset. The menu can contain as many as 99 selections and there may be more than one screen for each dataset. Four modes, ADD, CHANGE, SHOW, and DELETE are available for each menu number. ADD mode allows addition of new records to a dataset, CHANGE mode allows changes to existing records, SHOW mode allows the display of existing records but prevents any modifications, and DELETE mode allows the removal of records from the dataset.

Figure 13 illustrates the function keys available in the CHANGE mode. Only function key 2 is mode specific, the other functions are available in all modes. Three keys, find, forward browse, and backward browse locate records in a dataset. The find key locates a key item in a master dataset or the head of a chain in a detail dataset. The browse keys locate and allow movement forward or backward along a detail chain. Forward browse begins at the head of the chain and backward browse begins at the end of the chain. Browse works best along a sorted chain because the records will appear in a predictable order. The CHANGE, SHOW, and DELETE modes obviously require the record location keys but the keys are useful in the ADD mode also. Backward browse recalls to the screen the items from last record in a patient's chain and the operator can check that the information in previous items is consistent with the

new information. Items whose information hasn't changed from the last record are already present, thus the operator needs only to type items with new information and add a new record from the newly constructed screen.

As each record in a dataset is added, changed, or deleted, GSA provides a time-stamped print-out of the terminal screen and it puts the characters from the screen in a program buffer. The previous screen key can recall the information from the last screen quickly to provide the operator with another look at what was modified or to speed up the addition of a series of records with similar information. At any time the operator can print the current screen via the print screen key.

Limitations

GSA serves our clinical database needs very well but some enhancements could make it even more useful. Location of a record in a detail dataset is rapid only if it is located in a short sorted chain. Figure 9 contains an IMAGE/1000 schema optimal for use by GSA, INSIGHT, and QUERY reports. An automatic master dataset contains PTVIS, a concatenation of patient number and visit. The detail datasets GDTREA and GDEVAL now contain sorted chains to allow orderly browsing in GSA. The most useful chain is PTVIS because it enables the location of all of a patient's sites at a particular visit. Since there are never more than 4 sites in our hypothetical clinical study, it is not time-consuming to browse to a particular record. The next version of GSA will employ a new DIMENSION call that performs a chain read until a successful character match occurs between the screen and the database record. The new call directly locates an individual record.

Another limitation of VIEW (and therefore GSA) is screen size. VIEW will only accept screens up to about 47 lines in length. Many times the design of a 47 line screen providing an acceptable representation of a clinical report form is a difficult chore. Our terminals have at least 96 lines of screen memory and we would welcome the ability to use all of the lines available.

Techniques and Work-Arounds

Efficient use of the current version of GSA requires that detail datasets have at least one short sorted chain in order to locate records rapidly. Use of an automatic master dataset provides a way to accomplish this and short sorted chains also make access by other programs efficient and powerful.

PHASE 3: DBMAP (THE DATABASE MANIPULATOR)

Benefits

With QUERY, INSIGHT, and GSA we had partially fulfilled each of the requirements outlined in Figure 4, but there were still reporting tasks that required hand collation and calculation. For instance a report displaying the change in an estimated correction level from treatment to each evaluation required items from 2 datasets as did a report grouping treatment or evaluation parameters by sex. INSIGHT was able to provide some of these reports within the limitations listed above, but there were other reports such as those containing the calculation of days between dates that neither INSIGHT nor QUERY could produce. We designed DBMAP to enhance the power of QUERY and INSIGHT and to provide database tool to enable us to fulfill more fully the requirements listed in Figure 4.

Figure 10 contains a schema that is optimal for use by QUERY, INSIGHT, GSA, and DBMAP. DBMAP copies data from one dataset to another in response to instructions contained in an ASCII file produced with EDIT/1000. QUERY and INSIGHT then can operate on a single dataset that contains data collected from many datasets by DBMAP. DBMAP also works within a single dataset to produce an item such as IDNO from the concatenation of other items. In addition, DBMAP can recombine and transfer all the data from an old database to the datasets in a new redesigned database.

DBMAP examples

Figure 14 contains the DBMAP instructions to produce IDNO in the dataset DDHIST. The arguments in the PROCESS line are the process name, a name for a QUERY select-file, and a modification type. DBMAP can perform 5 types of modifications that differ in the way they add to or update records in the the output dataset. The types also differ in the way that they treat non-existent or duplicate link items. The DATA-BASE line contains the parameters database:security-code:cartridge, level-word, mode, and dataset. The FIND line is any legal QUERY find statement and it produces the select-file named in the PROCESS line. The FIND line is omitted if the named select-file references a file previously produced by QUERY. The LET lines instruct DBMAP to move characters from one item to another. LET IDNO = MDNO will cause DBMAP to copy all characters in the right item to the left item. LET IDNO:7:10 = PATNO will cause DBMAP to copy the characters 1 through 4 in PATNO into the positions 7 through 10 in IDNO. The process repeats the copies for all records in the select-file. The same DBMAP instruction file can contain other similar processes to fill IDNO in the DDTREA and DDEVAL datasets. Thus aided by DBMAP, INSIGHT can now produce a full patient report sorted by MDNO and the data entry operator did not need to type

redundant characters into IDNO nor did a programmer need to prepare a FORTRAN program such as the one in Figure 12.

Figure 15 contains DBMAP instructions to copy the contents of the SEX item in the DDHIST to an item named UTIL in the DDTREA dataset. The TO line contains the type of arguments in the DATA-BASE line plus a link argument. The link argument, this case PATNO, is an item list that corresponds to the argument of the LINK line. These 2 link arguments allow DBMAP to copy information from a record with a specific PATNO in DDHIST to all records with the same PATNO in DDTREA. We put 1 or more utility fields like UTIL in our datasets to provide space to copy data for special reporting purposes. UTILIN provides space for temporary integer data, although this is not strictly necessary because DBMAP will also perform data type conversions.

Figure 16 contains DBMAP instructions to compute the age of each patient on the day that DBMAP is run. The instructions LET AGE = @D #YEAR "\$T" - DOB tell DBMAP to perform a date function and fill the AGE item in the DDHIST dataset with the number of years between the contents of the DOB item and the system date "\$T". Any valid date item or literal date string in YYMMDD format can replace the \$T in a DBMAP date function. DBMAP can also compute days or months between dates and project a future dates from a date and a given interval.

Figure 17 illustrates how we use DBMAP to enable QUERY to count the number of patients at each visit in a dataset. Each record in the DDEVAL contains information about a particular site on a patient at a particular visit. The total number of records in the dataset therefore reflects the total number of evaluated sites at all evaluation visits. For clinical studies tracking and reporting purposes, it is important to know the number of patients that have been evaluated at each visit and without help neither QUERY nor INSIGHT can provide this information. The DATA-BASE line in Figure 17 has 3 additional arguments ptvis, \$F, siteno that specify operations performed on a sorted "pseudochain". DBMAP processes the records in the select-file in the sorted order of the concatenated items PTVIS and SITENO. The \$F after PTVIS instructs DBMAP to perform the LET operations in the process on a single record when the value of PTVIS changes. Thus, PTRECl is marked Y once for each patient at each visit. The additional sort by SITENO causes DBMAP to mark the patient's first site. Now, based on PTRECl, QUERY or INSIGHT can select a single record for each patient at each visit, thus counting each patient once at each visit.

Figure 18 illustrates that DBMAP can use items from different datasets to compute changes in patient parameters over time. The FROM line specifies the input dataset and its link item-list PATNO,VISIT,SITENO. The LINK line specifies

the link item-list PATNO,VISIT,SITENO for the dataset in the DATA-BASE line. DBMAP will match records from the DATA-BASE dataset and the FROM dataset based on the characters in each concatenated item-list and allow register computations with items in the input or output records. LET MDCRCH = @R instructs DBMAP to place the result of the computation in the MDCRCH item in the output dataset DDEVAL. The REGISTER line contains arguments, specified in Reverse Polish Notation, to compute the difference between MDCRTR in DDTEA and MDCREV in DDEVAL. DBMAP obtains the value of items from the output record by default, so MDTREA:::I instructs DBMAP to obtain the value of MDTREA from the input record in DDTEA instead of the output record in DDEVAL. This process also computes PTCRCH. There is no limit on the number of register operations that each process can contain.

Figure 19 contains DBMAP instructions that identify the occurrence of a treatment failure in a patient and then indicate that failure in all of that patient's records. For the hypothetical clinical study, a treatment failure occurs when both the patient and the doctor estimate that the correction level has decreased by 2. The FIND line in Figure 19 identifies these failures in the DDEVAL dataset and produces a select-file that identifies 1 or more sites on any patient for whom the treatment has failed. In this case, the TO dataset is the same as the DATA-BASE dataset and DBMAP fills FAILCH with an F for all records in the PATNO chain identified by each PATNO in the select-file. The argument, \$F, in the DATA-BASE line prevents redundant processing of the PATNO chain by instructing that DBMAP process the PATNO chain only once when it finds the first failure for a PATNO in the select-file.

Limitations

Although, the instructions for a simple DBMAP task, such as the one in Figure 14, are reasonably straight-forward, DBMAP has a cryptic command language for the specification of the more complex tasks such as the one in Figure 16. The DBMAP language represents a compromise between user-friendliness and ease of software development. There is another feature of DBMAP might be perceived as a limitation. DBMAP must run separately before a report generator, such as QUERY or INSIGHT, can report on the correctly processed data. This lengthens the time required to produce a report. We usually use CI command files to run DBMAP before report generation to guarantee correctly processed output listings.

Techniques

On the whole, DBMAP has proved to be a very powerful tool. It provides us with a special purpose instruction language to fulfill the database requirements for clinical studies

listed in Figure 4. DBMAP does lengthen the amount of time required to produce a report but the cost is not high for all the power that DBMAP provides. We compared processing times for QUERY, DBMAP, and INSIGHT during normal system activity. For 1000 records, QUERY required 5.7 minutes to sort a single dataset and produce a report file. For 1000 records, DBMAP required 8.5 minutes to perform a cross-dataset transfer and produce a log file, and for 1000 records, INSIGHT required 12.9 minutes to sort a single dataset and produce a report file. We have described only some of the simpler uses of DBMAP and we have found that we have been able to accomplish tasks of increasing complexity as we have become more proficient in its use.

PHASE 5: STATS - STATISTICS FOR IMAGE/1000

Benefits

We developed STATS to provide us with some simple scientific statistics that we could not obtain satisfactorily with QUERY or INSIGHT. We wanted to develop software that obtained its data directly from records identified by a QUERY select-file because then we could accomplish some statistical tasks rapidly by eliminating the need to produce, and reformat files for input to a separate statistics package. STATS provides 2 kinds of statistical reports which we will describe below.

STATS Examples

Figure 20 contains an example of the STATS instructions and the output for a BASIC report that provides count, mean, standard deviation, minimum, maximum, and standard error for grouped variables or ungrouped variables. The PROCESS line contains the process name, a name of a QUERY select-file, a level-word, mode, and the STATS keyword BASIC. The arguments in the DATA-BASE line are the database:security-code:cartridge, level-word, mode, dataset, and a group item list. The FIND line is any legal QUERY find statement and it produces the select-file named in the PROCESS line. The DATA-BASE and the FIND line may be omitted if the named select-file references a file previously produced by QUERY. If the GROUP line is not included, STATS performs the statistics based on all records referenced in the select-file. If the GROUP line is included, its argument must match the group item list in the DATA-BASE line and STATS will perform the statistics every time that the group item list changes value. STATS will process the all items referenced in the ITEM lines and the items may be individually scaled by multiplication or division and offset addition or subtraction. The offset can occur before or after the scaling. STATS will also exclude a single missing value per item. Figure 20 shows the items divided by 100 with a missing value specified as -999. At the end of the statistical report, STATS pro -

vides reference information that identifies the sources of the input file and the data items.

Figure 21 contains an example of the STATS instructions and the output for a FREQ report that provides row and column counts, totals, and percentages for 2 items. The PROCESS, DATA-BASE, FIND, and GROUP lines are similar to the lines in the BASIC instruction file except for the keyword FREQ in the PROCESS line. STATS will compute the frequencies of each value of the items in the ITEM lines unless an optional range list defines counting bins. In the range lists, parentheses indicate exclusion of an end point and square brackets indicate inclusion of an end point. Null values imply an infinite lower or upper bound. STATS will produce a row and column labeled 'other' if it encounters an item value that is not in the range list.

Limitations

STATS will only exclude 1 missing value per item. This is sometimes inconvenient because we have found that it is useful to code missing values based on the reason for their absence. We use DBMAP to recode such missing values before we process them with STATS. Another limitation is that STATS can only group data at one level based on the change of a concatenated item list e.g. MDNO and PATNO. This grouping is equivalent to the lowest group level in QUERY. Thus, one STATS report is needed to produce statistics grouped by MDNO and a second report is needed to produce statistics grouped by PATNO. Lastly, the STATS FREQ option can only group data into 20 bins. However, we have found that 20 bins are more than adequate for most tasks.

CONCLUSION

We have had our HP1000 for 2 and 1/2 years now to serve our clinical studies database needs and it has greatly enhanced our productivity at all phases. The excellent programming environment has enabled us to develop the general purpose software tools GSA, DBMAP, and STATS to aid and supplement HP'S IMAGE/1000 database and the database tools VIEW, DIMENSION, and INSIGHT from Polaris, Inc. By concentrating on the use and development of software tools instead of ad hoc applications programs, we now routinely fulfill the clinical studies requirements listed in Figure 4 without programming. Our programless solutions simplify documentation, greatly reduce the time required to implement a new database task, and they provide IMAGE/1000 with some relational capabilities making it easier for the user to enter and retrieve data in a more natural manner.

Collagen Scientific Computer System Hardware

HP1000

- RTE-A Operating System with VC+
- 2 Mbytes Memory
- 2 12040B 8 channel Mux cards
- 2 12005B Async cards
- 6 2392A Terminals
- 1 2631B Printer
- 1 2934 Printer
- 9 9816 Scientific Workstations
- 3 IBM PC clones
- 1 IBM XT

HP7214TD

- 132 Mbyte Disk
- 7970E 1600 BPI

HP7946A

- 55 Mbyte Disk
- Catridge Tape Drive

RS232 Distributed Data Switch from Metapath, Inc.

Figure 1

An Example of a Clinical Report Form

PPC/HA FOR THE REPAIR OF PERIODONTAL INTRAOSSEOUS DEFECTS

I. Demographic Data			
Patient Name: 		Patient ID: 	
Last First		Investigator ID: 	
Sex: M <input type="checkbox"/> F <input type="checkbox"/>		Investigator Name: 	
Date of Birth: 		Patient Racial Background:	
Mo Day Yr		<input type="checkbox"/> Asian <input type="checkbox"/> Black <input type="checkbox"/> Caucasian <input type="checkbox"/> Hispanic <input type="checkbox"/> Other Specify	
Social Security Number: 			
Address: 			
Telephone: 			
AC			
II. Patient History			
Part A: Patient History/Contraindications		<input type="checkbox"/> Medication/Food allergies (List):	
Check if applicable. IF YES, DO NOT ENROLL IN THE STUDY.			
<input type="checkbox"/> Lidocaine hypersensitivity <input type="checkbox"/> Current pregnancy <input type="checkbox"/> History of anaphylactoid reactions <input type="checkbox"/> Diabetes <input type="checkbox"/> Paget's disease <input type="checkbox"/> Osteoporosis <input type="checkbox"/> Osteomalacia <input type="checkbox"/> Cushing's Syndrome <input type="checkbox"/> Hyperthyroidism <input type="checkbox"/> Hyperparathyroidism <input type="checkbox"/> Chronic liver disease <input type="checkbox"/> None of the above		<input type="checkbox"/> Chemotherapy/radiotherapy If patient is currently <u>regularly</u> taking any of the following medications, check below. <input type="checkbox"/> Hydantoins (Dilantin) <input type="checkbox"/> Estrogen/birth control pills <input type="checkbox"/> Vitamin C in excess of 1 gm/day <input type="checkbox"/> Thyroid <input type="checkbox"/> ACTH <input type="checkbox"/> Antibiotics <input type="checkbox"/> None of the above	
If patient is currently <u>regularly</u> taking any of the following medications, check below. IF YES, DO NOT ENROLL IN THIS STUDY. <input type="checkbox"/> Nonsteroidal anti-inflammatories (aspirin) <input type="checkbox"/> Steroids <input type="checkbox"/> Anticoagulants (Coumadin, Heparin) <input type="checkbox"/> Tetracycline <input type="checkbox"/> Patient is not <u>regularly</u> taking any of the above		Part C: Patient History (check if applicable)	
Part B: Patient History Related to Protocol Precautions		<input type="checkbox"/> Autoimmune disease (Personal history) <input type="checkbox"/> Systemic or discoid lupus erythematosus <input type="checkbox"/> Rheumatoid arthritis <input type="checkbox"/> Polyarteritis nodosa <input type="checkbox"/> Hashimoto's thyroiditis <input type="checkbox"/> Graves' disease <input type="checkbox"/> Progressive systemic sclerosis (Scleroderma) <input type="checkbox"/> Dermatomyositis <input type="checkbox"/> Polymyositis <input type="checkbox"/> Psoriatic arthritis <input type="checkbox"/> Ulcerative colitis <input type="checkbox"/> Crohn's disease <input type="checkbox"/> Sjogren's disease <input type="checkbox"/> Reiter's disease <input type="checkbox"/> Mixed connective tissue disease <input type="checkbox"/> None of the above	
(Check if applicable and give date of most recent episode.) <input type="checkbox"/> Hayfever <input type="checkbox"/> Eczema <input type="checkbox"/> Asthma <input type="checkbox"/> Urticarial reactions or rashes 			
III. Test Information			
Date of Blood #1 Drawn 		<input type="checkbox"/> No blood drawn.	
Date of ZYDERM Skin Test 		Lot Number 	
Send this form, first blood specimen and signed Informed Consent Form together to Collagen Corporation.			
Investigator's Signature: 		Date: 	
		Mo Day Yr	
White and Yellow Copies: Send to Collagen Corporation Pink Copy: Retain for patient record (1066pro p.1)			

Figure 2

Schematic Representation of a Hypothetical Clinical Study

MD
Enrollment
Form

Enrollment
MDNO

Patient
History
Form

History Form
PATNO

Patient
Treatment
Forms

Treat Form
PATNO
Sites 1...n

Treat 1

Treat 2

Treat 3

Patient
Evaluation
Forms

Eval Form
PATNO
Sites 1...n

Eval D42

Eval M03

Eval M06

Eval Y01

Eval Y02

Figure 3

Database Requirements for Clinical Studies

- 1) Storage, verification, and selection of patient parameters provided on each clinical report form.
- 2) Selection of data groups based on time point, patient, and site of treatment.
- 3) The ability to add new parameters to the database during the course of a study.
- 4) Retrieval of full patient reports compiled from all data on each patient.
- 5) Selection of data for scientific analysis based on flexible user-defined criteria.
- 6) Computation of changes in patient parameters over time.
- 7) Identification of complete cases under a specific criterion.
- 8) Identification of treatment successes and failures.

Figure 4

General Database Design Considerations for Optimal Use of Database Tools

Tool	+ Benefit - Limitation	Considerations
QUERY (HP)	+ + + + - - - - -	implementation is quick and simple select and report commands are straight-forward data modification and report commands are available at the same time can name select files for subsequent use cannot change sort or key items the only user prompt is the item name items are presented in schema order reports are limited to a single dataset reports contain only 10 lines no arithmetic operations
INSIGHT (Polaris)	+ + + + - - - - -	screen oriented user prompts extensive reports from multiple datasets reports can contain 66 lines register and arithmetic operations data modification commands not available cannot name select files multiple dataset connector or link items must be the same lengths and types links and sorts must be whole items link item must be the highest sort item higher group breaks force lower breaks and limit group operations on linked datasets
BSA (Collagen)	+	no programming required for screen database implementation
VIEW and DIMENSION (Polaris)	+ + + + -	database access is menu driven screens can contain any user prompts items are presented in any order can change sort or key items implementation slower than QUERY because of screen design
DBMAP (Collagen)	+ + + + -	takes advantage of sorted chains moves data between databases and datasets links can be multi-item of different lengths and types can recombine data so the power of any report generator is enhanced and redundant data entry is reduced DBMAP must run separately after data entry and before report generation

Figure 5

Specific Database Design Elements for Optimal Use of Database Tools

Tool	Design Element
QUERY	No sort items because an update of a sort item requires deletion and re-entry of an entire record.
INSIGHT	Link items were added to enhance for cross-dataset reporting.
GSA using VIEW and DIMENSION	An automatic master dataset containing record identification items was added to provide for rapid screen access during finds and updates.
DEMAP	Utility items were added to datasets to contain computed and temporary find, sort, link, and marker items.

Figure 6

IMAGE/1000 Database Schema Optimal for Use by QUERY

```

sets:          << for optimal use by QUERY          >>
<<                                                    >>
  name:qmdm::23,m;
  entry:  << one record per md                      >>
          mdno(2),      << x6      md number        >>
          mdname,       << x16   md last name        >>
          mdspec,       << x2    md speciality D = dermatologist >>
          << P = plastic surgeon E = eye/ear/nose/throat >>
  capacity:5;      << total md's                    >>
<<                                                    >>
  name:qdhist::23,m;
  entry:  << one record per each patient history    >>
          patno(2),     << x4      patient id number >>
          custno,       << x6      md number         >>
          date,         << x6      date history taken (yyymmdd) >>
          patnm,        << x26     patient name,      (last, first) >>
          dob,          << x6      date of birth      (yyymmdd) >>
          yob,          << i1      year of birth       >>
          sex,          << x2      sex                (M/F)       >>
  capacity:11;      << total patients                >>
<<                                                    >>
  name:qdtrea::23,d;
  entry:  << one record per site per patient treatment >>
          patno (qdhist), << x4      patient number    >>
          mdno (qmdm),   << x6      md number         >>
          visit,         << x4      treatment number    (1/2/3)    >>
          date,          << x6      treatment date      (yyymmdd) >>
          siteno,        << x2      site number          (G1/G2/G3/Z1/Z2/Z3) >>
          site,          << x4      site code            (GL/NL)     >>
          matl,          << x2      material used        (G/Z)       >>
          vol,           << i1      volume              (50/150/1050)(0-10ML) >>
          mdcrr,         << i1      estimates of correction (1 worst - 5 best) >>
          ptcrr,         << i1      after treatment     by md and patient >>
          ptrecl,        << x2      patient's first site ? (Y/ )    >>
  capacity:99;      << 11 patients * 3 treatments * 3 sites    >>
<<                                                    >>
  name:qdeval::23,d;
  entry:  << one record per site per patient evaluation >>
          patno (qdhist), << x4      patient number    >>
          mdno (qmdm),   << x6      md number         >>
          visit,         << x4      visit codes sort in time D42 = 7 weeks >>
          << M03 = 3 months M06 = 6 months >>
          << Y01 = 1 year Y02 = 2 years >>
          date,          << x6      evaluation date      (yyymmdd) >>
          daycnt,        << i1      days since treatment >>
          siteno,        << x2      site number          (G1,G2,G3,Z1,Z2,Z3) >>
          site,          << x4      site code            (NL,GL,CHK,CHN,GRAL) >>
          mdcrr,         << i1      estimates of correction (1 worst - 5 best) >>
          ptcrr,         << i1      at this evaluation by md and patient >>
          ptrecl,        << x2      patient's first site ? (Y/ )    >>
  capacity:165;     << 11 patients * 5 evaluations * 3 sites    >>

```

Figure 7

IMAGE/1000 Database Schema Optimal for Use by QUERY and INSIGHT

```

sets:          << for optimal use by QUERY and INSIGHT          >>
<<
  name:idmd::23,m; << ensures unique mdno in idmd dataset      >>
  entry: << one record per md                                   >>
    mdno(2), << x6 md number                                     >>
    mdname, << x16 md last name                                   >>
    mdspec, << x2 md speciality D = dermatologist              >>
    << P = plastic surgeon E = eye/ear/nose/throat             >>
  capacity:5; << total md's                                     >>
<<
  name:idhist::23,m; << ensures unique patno in idhist dataset  >>
  entry: << one record per each patient history                >>
    patno(2), << x4 patient id number                           >>
    idno, << x10 patno & mdno for INSIGHT reports>>
    custno, << x6 md number                                       >>
    date, << x6 date history taken (yyymmdd)                    >>
    patnm, << x26 patient name, (last, first)                   >>
    dob, << x6 date of birth (yyymmdd)                          >>
    yob, << i1 year of birth                                     >>
    sex, << x2 sex (M/F)                                         >>
  capacity:11; << total patients                                 >>
<<
  name:idtrea::23,d;
  entry: << one record per site per patient treatment          >>
    patno (idhist), << x4 patient number                        >>
    idno, << x10 patno & mdno for INSIGHT reports>>
    mdno (idmd), << x6 md number                                 >>
    visit, << x4 treatment number (1/2/3)                       >>
    date, << x6 treatment date (yyymmdd)                        >>
    siteno, << x2 site number (G1/G2/G3/Z1/Z2/Z3)               >>
    site, << x4 site code (GL/NL)                                >>
    matl, << x2 material used (G/Z)                              >>
    vol, << i1 volume (50/150/1050)(0-10ML)                     >>
    mdcrr, << i1 estimates of correction (1 worst - 5 best)     >>
    ptcrr, << i1 after treatment by md and patient              >>
    ptrecl, << x2 patient's first site ? (Y/ )                  >>
  capacity:99; << 11 patients * 3 treatments * 3 sites         >>
<<
  name:ideval::23,d;
  entry: << one record per site per patient evaluation          >>
    patno (idhist), << x4 patient number                        >>
    idno, << x10 patno & mdno for INSIGHT reports>>
    mdno (idmd), << x6 md number                                 >>
    visit, << x4 visit codes sort in time D42 = 7 weeks         >>
    << M03 = 3 months M06 = 6 months                             >>
    << Y01 = 1 year Y02 = 2 years                               >>
    date, << x6 evaluation date (yyymmdd)                       >>
    daycnt, << i1 days since treatment                           >>
    siteno, << x2 site number (G1,G2,G3,Z1,Z2,Z3)               >>
    site, << x4 site code (NL,GL,CHK,CHN,ORAL)                  >>
    mdcrr, << i1 estimates of correction (1 worst - 5 best)     >>
    ptcrr, << i1 at this evaluation by md and patient           >>
    ptrecl, << x2 patient's first site ? (Y/ )                  >>
  capacity:165; << 11 patients * 5 evaluations * 3 sites       >>

```

Figure 8

IMAGE/1000 Database Schema Optimal for Use by QUERY, INSIGHT, and GSA

```

sets:          << for optimal use by QUERY, INSIGHT, and GSA          >>
<<
  name:gdptvi::23,a; << to provide for rapid screen find              >>
  entry: << one record per site per treatment or evaluation          >>
        ptvis(2), << x8 patno&visit for GSA finds>>
        capacity:307; << 99 treats + 165 evals + some extra          >>
<<
  name:gdmd::23,m; << ensures unique mdno in dataset                  >>
  entry: << one record per md                                          >>
        mdno(2), << x6 md number                                       >>
        mdname, << x16 md last name                                    >>
        mdspec, << x2 md speciality D = dermatologist                 >>
        << P = plastic surgeon E = eye/ear/nose/throat                >>
        capacity:5; << total md's                                     >>
<<
  name:gdhist::23,m; << patno chain is sorted in time by visits      >>
  entry: << one record per each patient history                      >>
        patno(2), << x4 patient id number                             >>
        idno, << x10 patno & mdno for INSIGHT reports>>
        custno, << x6 md number                                       >>
        date, << x6 date history taken (yyymmdd)                     >>
        patnm, << x26 patient name, (last, first)                   >>
        dob, << x6 date of birth (yyymmdd)                           >>
        yob, << i1 year of birth                                     >>
        sex, << x2 sex (M/F)                                         >>
        capacity:11; << total patients                                >>
<<
  name:gdtrea::23,d;
  entry: << one record per site per patient treatment              >>
        ptvis (gdptvi(siteno)), << x8 patno&visit for GSA finds>>
        patno (gdhist(visit)), << x4 patient number                 >>
        idno, << x10 patno & mdno for INSIGHT reports>>
        mdno (gdmd (patno)), << x6 md number                         >>
        visit, << x4 treatment number (1/2/3)                       >>
        date, << x6 treatment date (yyymmdd)                       >>
        siteno, << x2 site number (G1/G2/G3/Z1/Z2/Z3)               >>
        site, << x4 site code (GL/NL)                               >>
        matl, << x2 material used (G/Z)                             >>
        vol, << i1 volume (50/150/1050)(0-10ML)                   >>
        mdcrr, << i1 estimates of correction (1 worst - 5 best)     >>
        ptcrr, << i1 after treatment by md and patient              >>
        ptrecl, << x2 patient's first site ? (Y/ )                 >>
        capacity:99; << 11 patients * 3 treatments * 3 sites        >>
<<
  name:gdeval::23,d;
  entry: << one record per site per patient evaluation              >>
        ptvis (gdptvi(siteno)), << x8 patno&visit for GSA finds>>
        patno (gdhist(visit)), << x4 patient number                 >>
        mdno (gdmd (patno)), << x6 md number                         >>
        visit, << x4 visit codes sort in time D42 = 7 weeks         >>
        << M03 = 3 months M06 = 6 months                             >>
        << Y01 = 1 year Y02 = 2 years                                >>
        date, << x6 evaluation date (yyymmdd)                       >>
        siteno, << x2 site number (G1,G2,G3,Z1,Z2,Z3)               >>
        site, << x4 site code (NL,GL,CHK,CHN,ORAL)                 >>
        mdcrr, << i1 estimates of correction (1 worst - 5 best)     >>
        ptcrr, << i1 at this evaluation by md and patient            >>
        ptrecl, << x2 patient's first site ? (Y/ )                 >>
        daycnt, << i1 days since treatment                          >>
        idno, << x10 patno & mdno for INSIGHT reports>>
        capacity:165; << 11 patients * 5 evaluations * 3 sites      >>

```

Figure 9

IMAGE/1000 Database Schema Optimal
for Use by QUERY, INSIGHT, GSA, and DBMAP

```

sets:      << for optimal use by QUERY, INSIGHT, GSA, and DBMAP
(<<
  name:ddptvis::23,a; << to provide for rapid screen find
  entry: << one record per site per treatment or evaluation
    ptvis(2), << x8 patno&visit for GSA finds
  capacity:307; << 99 treats + 165 evals + some extra
(<<
  name:ddmd::23,m; << ensures unique mdno in dataset
  entry: << one record per md
    mdno(2), << x6 md number
    mdname, << x16 md last name
    mdspec, << x2 md speciality D = dermatologist
    << P = plastic surgeon E = eye/ear/nose/throat
  capacity:5; << total md's
(<<
  name:ddhist::23,m; << patno chain is sorted in time by visits
  entry: << one record per each patient history
    patno(2), << x4 patient id number
    custno, << x6 md number
    date, << x6 date history taken (yyymmdd)
    patnm, << x26 patient name, (last, first)
    dob, << x6 date of birth (yyymmdd)
    sex, << x2 sex (M/F)
    <<
    << items below loaded or computed by DBMAP
    <<
    idno, << x10 mdno&patno for INSIGHT reports
    age, << i1 age of patient
    util, << x30 utility fields
    utilin, << i1 utility fields
  capacity:11; << total patients
(<<
  name:ddtrea::23,d;
  entry: << one record per site per patient treatment
    ptvis (ddptvis(siteno)), << x8 patno&visit for GSA finds
    patno (ddhist(visit)), << x4 patient number
    mdno (ddmd (patno)), << x6 md number
    visit, << x4 treatment number (1/2/3)
    date, << x6 treatment date (yyymmdd)
    siteno, << x2 site number (G1/G2/G3/Z1/Z2/Z3)
    site, << x4 site code (GL/HL)
    matl, << x2 material used (GZ)
    vol, << i1 volume (50/150/1050) (0-10ML)
    mdcrrr, << i1 estimates of correction (1 worst - 5 best)
    ptcrrr, << i1 after treatment by md and patient
    <<
    << items below loaded or computed by DBMAP
    <<
    idno, << x10 patno & mdno for INSIGHT reports
    ptrecl, << x2 patient's first site ? (Y/ )
    util, << x30 utility fields
    utilin, << i1 utility fields
  capacity:99; << 11 patients * 3 treatments * 3 sites
(<<
  name:dddeval::23,d;
  entry: << one record per site per patient evaluation
    ptvis (ddptvis(siteno)), << x8 patno&visit for GSA finds
    patno (ddhist(visit)), << x4 patient number
    mdno (ddmd (patno)), << x6 md number
    visit, << x4 visit codes sort in time D42 = 7 weeks
    << M03 = 3 months M06 = 6 months
    << Y01 = 1 year Y02 = 2 years
    date, << x6 evaluation date (yyymmdd)
    siteno, << x2 site number (G1,G2,G3,Z1,Z2,Z3)
    site, << x4 site code (NL,GL,CHK,CHN,ORAL)
    mdcrrv, << i1 estimates of correction (1 worst - 5 best)
    ptcrrv, << i1 at this evaluation by md and patient
    <<
    << items below loaded or computed by DBMAP
    <<
    << change in estimate eg mdcrr - mdcrrr
    mdcrrch, << i1 change in md estimate
    ptcrrch, << i1 change in patient estimate
    ptrecl, << x2 patient's first site ? (Y/ )
    daycnt, << i1 days since treatment
    failrc, << x2 treatment failure in record
    failch, << x2 treatment failure in pseudo chain
    idno, << x10 patno & mdno for INSIGHT reports
    util, << x30 utility fields
    utilin, << i1 utility fields
  capacity:165; << 11 patients * 5 evaluations x 3 sites

```

Figure 10

Effects of the INSIGHT Connector Item on Cross-Dataset Grouping

If PATNO is the connector item:

PATNO 1	MDNO 1	HISTORY
PATNO 1	MDNO 1	TREAT 1
PATNO 1	MDNO 1	EVAL 1
PATNO 1	MDNO 1	EVAL 2
PATNO 2	MDNO 5	HISTORY
PATNO 2	MDNO 5	TREAT 1
PATNO 2	MDNO 5	EVAL 1
PATNO 2	MDNO 5	EVAL 2
PATNO 3	MDNO 1	HISTORY
PATNO 3	MDNO 1	TREAT 1
PATNO 3	MDNO 1	EVAL 1
PATNO 3	MDNO 1	EVAL 2
PATNO 4	MDNO 5	HISTORY
PATNO 4	MDNO 5	TREAT 1
PATNO 4	MDNO 5	EVAL 1
PATNO 4	MDNO 5	EVAL 2

If MDNO is the connector item:

PATNO 1	MDNO 1	HISTORY
PATNO 3	MDNO 1	HISTORY
PATNO 1	MDNO 1	TREAT 1
PATNO 3	MDNO 1	TREAT 1
PATNO 1	MDNO 1	EVAL 1
PATNO 1	MDNO 1	EVAL 2
PATNO 3	MDNO 5	HISTORY
PATNO 4	MDNO 5	HISTORY
PATNO 2	MDNO 5	TREAT 1
PATNO 4	MDNO 5	TREAT 1
PATNO 2	MDNO 5	EVAL 1
PATNO 2	MDNO 5	EVAL 2
PATNO 4	MDNO 5	EVAL 1
PATNO 4	MDNO 5	EVAL 2

If IDNO is the connector item:

IDNO 11	PATNO 1	MDNO 1	HISTORY
IDNO 11	PATNO 1	MDNO 1	TREAT 1
IDNO 11	PATNO 1	MDNO 1	EVAL 1
IDNO 11	PATNO 1	MDNO 1	EVAL 2
IDNO 13	PATNO 3	MDNO 1	HISTORY
IDNO 13	PATNO 3	MDNO 1	TREAT 1
IDNO 13	PATNO 3	MDNO 1	EVAL 1
IDNO 13	PATNO 3	MDNO 1	EVAL 2
IDNO 52	PATNO 2	MDNO 5	HISTORY
IDNO 52	PATNO 2	MDNO 5	TREAT 1
IDNO 52	PATNO 2	MDNO 5	EVAL 1
IDNO 52	PATNO 2	MDNO 5	EVAL 2
IDNO 54	PATNO 4	MDNO 5	HISTORY
IDNO 54	PATNO 4	MDNO 5	TREAT 1
IDNO 54	PATNO 4	MDNO 5	EVAL 1
IDNO 54	PATNO 4	MDNO 5	EVAL 2

Figure 11

A Fortran Program to Concatenate 2 Items in a Dataset

```

PROGRAM COCAT
  INTEGER Ibuf(40),Tlog,Inbuf(40)
  CHARACTER Cbuf*80,Cinbuf*80
  EQUIVALENCE (Cbuf,Ibuf),(Cinbuf,Inbuf)
  INTEGER Dbase(9),Dset(3),Level(3),Mode
  INTEGER Out(3),In1(3),In2(3),Inlength,Istat(10)
  LOGICAL End_of_file
  *****
  *
  *   get run string and parse into Data-base Data-set Level Mode *
  *               Destination-item and two Source-items *
  *
  *****
  CALL Getst(Ibuf,40,Tlog)
  CALL ParseString(Cbuf,Dbase,Dset,Level,Mode,Out,In1,In2)
  CALL Dbopn(Dbase,Level,Mode,Istat)
  IF (Istat.ne.0) GOTO 9999
  *****
  *
  *   a) get value of the first source item
  *   b) get value of the second source item
  *   c) concatenate these values
  *   d) put value of the concatenated string in destination item
  *
  *****
  CALL Dbget(Dbase,Dset,2,Istat,In1,Ibuf) ! read first, get In1
  IF (Istat.ne.0.and.Istat.ne.12) GOTO 9999
  Inlength=Istat(2)*2
  End_of_file=(Istat.eq.12)
  DO WHILE (.not.End_of_file)
    CALL Dbget(Dbase,Dset,1,Istat,In2,Inbuf) ! re-read, get In2
    IF (Istat.ne.0) GOTO 9999
    Cbuf(Inlength+1:)=Cinbuf ! concatenate
    CALL Dbldc(Dbase,Dset,1,Istat)
    IF (Istat.ne.0) GOTO 9999
    CALL Dbupd(Dbase,Dset,1,Istat,Out,Ibuf) ! put in the destination
    IF (Istat.ne.0) GOTO 9999
    CALL Dbunl(Dbase,Dset,1,Istat)
    IF (Istat.ne.0) GOTO 9999
    CALL Dbget(Dbase,Dset,2,Istat,In1,Ibuf) ! read next, get In1
    IF (Istat.ne.0.and.Istat.ne.12) GOTO 9999
    End_of_file=(Istat.eq.12)
  END DO
  CALL Dbcls(Dbase,Dset,1,Istat)
  STOP 'Successful'
9999 WRITE(1,*) 'Image error ',Istat(1),'. Abnormal ending.
  CALL Dbcls(Dbase,Dset,1,Istat)
  END

SUBROUTINE ParseString(String,Dbase,Dset,Level,Mode,Out,In1,In2)
  CHARACTER String*(*)
  INTEGER Dbase(*),Dset(*),Level(*),Mode,Out(*),In1(*),In2(*)
  INTEGER Ibuffer(40),DecimalToInt,ierr
  CHARACTER Cbuffer*80
  EQUIVALENCE (Cbuffer,Ibuffer)
  Dbase=2H
  CALL SplitString(String,Cbuffer,String)
  CALL Smove(Ibuffer,1,16,Dbase,3)
  CALL SplitString(String,Cbuffer,String)
  CALL Smove(Ibuffer,1,6,Dset,1)
  CALL SplitString(String,Cbuffer,String)
  CALL Smove(Ibuffer,1,6,Level,1)
  CALL SplitString(String,Cbuffer,String)
  Mode=DecimalToInt(Cbuffer,ierr)
  CALL SplitString(String,Cbuffer,String)
  CALL Smove(Ibuffer,1,6,Out,1)
  CALL SplitString(String,Cbuffer,String)
  CALL Smove(Ibuffer,1,6,In1,1)
  CALL SplitString(String,Cbuffer,String)
  CALL Smove(Ibuffer,1,6,In2,1)
  RETURN
END

```

Figure 12

GSA - Generic Screen Access

Top Menu

1 - Screen 1 for Dataset 1
2 - Screen 2 for Dataset 1
3 - Screen 1 for Dataset 2
4 - Screen 1 for Dataset 3

2 Type menu # and select
mode key below

Add	Change	Show	Delete	Exit System			
-----	--------	------	--------	----------------	--	--	--

Screen 2 for Dataset 1

Item 1	Item 2	Item 3	Item 4
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
Item 5		Item 6	Item 7
<input type="text"/>		<input type="text"/>	<input type="text"/>

CHANGE press f2 key to change record **CHANGE**

find	CHANGE	forward browse	backward browse	previous screen	print screen		Exit
------	--------	-------------------	--------------------	--------------------	-----------------	--	------

(ADD, CHANGE, SHOW, and DELETE modes have similar f keys)

Copyright 1986 Collagen Corp.

Figure 13

Task: Fill IDNO with the concatenation of MDNO and PATNO

DBMAP instructions for 1 dataset:

```
PROCESS: idno_demo, qyidno_demo, 2
  DATA-BASE ddemdb:7:23, write, 1, ddhist
  FIND ddhist.patno ine " " end
  LET idno = mdno
  LET idno:7:10 = patno
END
```

Figure 14

Task: Copy SEX from DDHIST to UTIL in DDTREA

DBMAP instructions:

```
PROCESS: sex_demo, qysex_demo, 2
  DATA-BASE ddemdb:7:23, read, 1, ddhist
  FIND ddhist.patno ine " " end
  TO ddemodb:7:23, write, 1, ddtrea, patno
  LINK patno
  LET util = sex
END
```

Figure 15

Task: Compute AGE today from year of birth YOB

DBMAP instructions:

```
PROCESS: age_demo, qyage_demo, 2
  DATA-BASE ddemdb:7:23, write, 1, ddhist
  FIND ddhist.patno ine " " end
  LET age = @D #YEAR "$T" - dob
END
```

Figure 16

Task: Fill PTREC1 with Y if record the occurs first in
a sorted PTVIS chain

DBMAP instructions for 1 dataset:

```
PROCESS:chain_demo,qychain_demo,2
  DATA-BASE ddemdb:7:23,write,1,ddtrea,ptvis,$F,siten0
  FIND ddtrea.patno ine " " end
  LET ptrec1 = "Y"
END
```

Figure 17

Task: Fill MDCRCH and PTCRCH in DDEVAL with MDCREV in
DDEVAL minus MDCRTR in DDTREA, ditto for PT.

DBMAP instructions:

```
PROCESS:xdschange_demo,qyxdschange_demo,2
  DATA-BASE ddemdb:7:23,read,1,ddeval
  FIND ddeval.patno ine " " end
  FROM ddemdb:7:23,write,1,ddtrea,patno,visit,siten0
  LINK patno,visit,siten0
  LET mdcrch = @R
  REGISTER mdcrev mdcrttr::::l -
  LET ptcrch = @R
  REGISTER ptcrev ptcrttr::::l -
END
```

Figure 18

Task: Find site failures in DDEVAL where MDCRCH and PTCRCH
are < -2. Fill failch with F for all patient records if
any site at any visit failed.

DBMAP instructions:

```
PROCESS:mark_failure,qymark_failure,2
  DATA-BASE ddemdb:7:23,write,1,ddeval,patno,$F
  FIND ddeval.mdcrch ilt "-2" and ptcrch ilt "-2" end
  TO ddemdb:7:23,write,1,ddeval,patno
  LINK patno
  LET failch = "F"
END
```

Figure 19

Instructions and Output for a STATS BASIC Process

```
PROCESS:evaluation&change qyevaluation read 1 BASIC
DATA-BASE demmdb:2:23 read 1 ddeval visit
FIND ddeval.patno ine " " end
GROUP visit
ITEM daycnt / 100 miss -999
ITEM mdcrev / 100 miss -999
ITEM ptcrev / 100 miss -999
ITEM mdcrch / 100 miss -999
ITEM ptcrch / 100 miss -999
```

END

VISIT

D00

ITEM	Count	Mean	Std Dev	Minimum	Maximum	Std Err
DAYCNT	77	16.273	5.513	7.000	31.000	.628
MDCREV	77	19.636	4.239	11.000	29.000	.483
PTCREV	77	17.273	6.114	5.000	33.000	.697
MDCRCH	77	15.273	5.101	8.000	30.000	.581
PTCRCH	77	16.494	3.865	9.000	30.000	.440

VISIT

D42

ITEM	Count	Mean	Std Dev	Minimum	Maximum	Std Err
DAYCNT	77	21.305	5.218	0.000	37.500	.595
MDCREV	76	23.816	4.127	16.000	34.000	.473
PTCREV	76	22.684	4.894	14.000	40.000	.561
MDCRCH	76	20.487	4.577	13.000	35.000	.525
PTCRCH	76	20.368	3.895	14.000	34.000	.447

VISIT

M03

ITEM	Count	Mean	Std Dev	Minimum	Maximum	Std Err
DAYCNT	76	20.645	5.209	0.000	37.500	.598
MDCREV	75	23.227	3.954	16.000	32.000	.457
PTCREV	75	21.947	4.879	14.000	40.000	.563
MDCRCH	75	19.893	4.599	12.000	35.000	.531
PTCRCH	75	19.707	3.941	14.000	34.000	.455

VISIT

ALL

ITEM	Count	Mean	Std Dev	Minimum	Maximum	Std Err
DAYCNT	230	19.402	5.748	0.000	37.500	.379
MDCREV	228	22.211	4.494	11.000	34.000	.298
PTCREV	228	20.614	5.831	5.000	40.000	.386
MDCRCH	228	18.531	5.293	8.000	35.000	.351
PTCRCH	228	18.842	4.240	9.000	34.000	.282

*** Reference ***

```
Input file : INTEREX.STAT
Process    : EVALUATION&CHANGE
Select file : /FORT/TU/QYEVALUATION:::1:6:128
Data base  : DDEMDB:2:23      Set DDEVAL
```

Figure 20

Instructions and Output for a STATS FREQ Process

```
PROCESS:sex_age qysex_age read 1 FREQ
DATA-BASE ddemdb:2:23 read 1 ddhist mdno
FIND ddhist.patno ine " " end
GROUP mdno
ITEM age (:201 [21:40] [41:60] [61:])
ITEM sex
END
```

MDNO
100058

SEX	MIN	>= 21	>= 41	>= 61	Total
	<= 20	<= 40	<= 60	MAX	
F	ln = 3 I% = 18.75 lr% = 27.27 lc% = 75.00	ln = 7 I% = 43.75 lr% = 63.64 lc% = 77.78	ln = 1 I% = 6.25 lr% = 9.09 lc% = 33.33	ln = 0 I% = 0.00 lr% = 0.00 lc% = *****	ln = 11 I% = 68.75
M	ln = 1 I% = 6.25 lr% = 20.00 lc% = 25.00	ln = 2 I% = 12.50 lr% = 40.00 lc% = 22.22	ln = 2 I% = 12.50 lr% = 40.00 lc% = 66.67	ln = 0 I% = 0.00 lr% = 0.00 lc% = *****	ln = 5 I% = 31.25
Total of Column	ln = 4 I% = 25.00	ln = 9 I% = 56.25	ln = 3 I% = 18.75	ln = 0 I% = 0.00	ln = 16

MDNO
560215

SEX	MIN	>= 21	>= 41	>= 61	Total
	<= 20	<= 40	<= 60	MAX	
F	ln = 3 I% = 12.50 lr% = 17.65 lc% = 60.00	ln = 4 I% = 16.67 lr% = 23.53 lc% = 66.67	ln = 10 I% = 41.67 lr% = 58.82 lc% = 76.92	ln = 0 I% = 0.00 lr% = 0.00 lc% = *****	ln = 17 I% = 70.83
M	ln = 2 I% = 8.33 lr% = 28.57 lc% = 40.00	ln = 2 I% = 8.33 lr% = 28.57 lc% = 33.33	ln = 3 I% = 12.50 lr% = 42.86 lc% = 23.08	ln = 0 I% = 0.00 lr% = 0.00 lc% = *****	ln = 7 I% = 29.17
Total of Column	ln = 5 I% = 20.83	ln = 6 I% = 25.00	ln = 13 I% = 54.17	ln = 0 I% = 0.00	ln = 24

MDNO
ALL

SEX	MIN	>= 21	>= 41	>= 61	Total
	<= 20	<= 40	<= 60	MAX	
F	ln = 6 I% = 15.00 lr% = 21.43 lc% = 66.67	ln = 11 I% = 27.50 lr% = 39.29 lc% = 73.33	ln = 11 I% = 27.50 lr% = 39.29 lc% = 68.75	ln = 0 I% = 0.00 lr% = 0.00 lc% = *****	ln = 28 I% = 70.00
M	ln = 3 I% = 7.50 lr% = 25.00 lc% = 33.33	ln = 4 I% = 10.00 lr% = 33.33 lc% = 26.67	ln = 5 I% = 12.50 lr% = 41.67 lc% = 31.25	ln = 0 I% = 0.00 lr% = 0.00 lc% = *****	ln = 12 I% = 30.00
Total of Column	ln = 9 I% = 22.50	ln = 15 I% = 37.50	ln = 16 I% = 40.00	ln = 0 I% = 0.00	ln = 40

```
*** Reference ***
Input file : INTEREX.FREQ
Process    : SEX_AGE
Select file : /FORT/TU/QYSEX_AGE::1:6:129
Data base  : DDEMDB:2:23      Set DDHIST
```

Figure 21

CC Word/100 — HP1000 Word Processing

by: Behrens, Jens

We regret that this paper
was not received for
inclusion in these proceedings.

Office Automation in an HP1000 Environment

by: Destra, Theresa

We regret that this paper
was not received for
inclusion in these proceedings.

HOW TO CHOOSE AN INSTRUMENT CONTROLLER:

A TUTORIAL

Terie Robinson
Hewlett-Packard Co.
3003 Scott Blvd.
Santa Clara, CA 95054

INTRODUCTION

The purpose behind this paper is to help people understand why "just any ol' computer" will not necessarily solve a given control problem. A particular computer may be ineffective, even if it has an assortment of accessories that seem workable. There is a lot more to the job of instrument control than it would first appear.

In that vein, we'll first take a look at the instrument control issue in terms of the measurement control task. Out of this will come a list of questions which must be asked about a given computer. These questions can be used to ascertain its ability to perform as an instrument controller.

Fundamental control capability is just part of the problem. There are features of the particular application which will impose additional requirements. For example, the measurement data may have to be input very quickly or very slowly. Needs such as this will further disqualify potential instrument controllers from a particular application.

The physical location of the system, and certain implementation concerns must be explored to complete the requirements list.

I'd like to point out that one of the major problems with topic of instrument control is that people throw terms around without defining them. This has led to a certain amount of confusion. For purposes of clarity, I'll establish some working definitions for the most important terms. A more complete list of terms and their definitions can be found in the glossary at the end of the paper.

TERMS

The first thing that should be discussed is what to call this computer application area? Many in the industry call it **data acquisition**, others, **real-time**. The first implies that the user of the phrase is looking at the problem from the instrument perspective (acquiring physical data), while the latter is looking at the problem from the computer end (communicating with devices in the outside world in a timely manner). There is also **automated test** and, a term that has been brought recently into general use within Hewlett-Packard: **measurement automation**. (Perhaps these would indicate an attempt to look at the problem from both sides at once.)

An **instrument controller** is a computer which controls the operation of instruments. We might call it a **controller**, for short. Industry-wide, other terms for an instrument controller include **data acquisition computer** and **real-time computer**.

In order to be a controller, a computer must have both hardware and software to perform I/O, so as to communicate with instrumentation. An **interface card** provides the hardware necessary to convert from the world of the computer to the world of the instrument. A piece of software, called an **interface driver**, is used by the operating system to control the interface.

Measurements performed by an instrument often require immediate transfer to the controller. However, the controller may have other things to do while the measurement is being made. Interrupts are a hardware means of signalling the controller for service. Thus, an instrument can interrupt the CPU via the interface.

However, if the operating system is unable to deal with the interrupt as an event, not much can be done to override that inability. Therefore, an operating system with that capability might be desirable. Labelled a **real-time system**,

it's designed to respond to a physical event in device-determined time frames.

Real-time therefore connotes a fast response to interrupts. The time it should take to respond has not been quantified by the industry, i.e. response time has no set value associated with it. Its required value can vary *radically*, from application to application.

Real-time also suggests an interrupt-driven, multi-tasking operating system. This means that it has the capability to run more than one user program at the same time. If an interrupt is detected, it will drop everything to take care of the event. RTE (Real Time Executive operating system) on the HP 1000 mini-computers works this way. Programs designated as "real-time" in RTE will not "share" the CPU unless it asks for a slow I/O transaction or access to a system resource which has been locked by another user-level program. At this time, the program will be suspended until the request has been satisfied. Certain other I/O requests will not suspend the program.

A real-time system contrasts with a time-sharing system, which uses time slices to determine how long a particular program can use the CPU. Hardware interrupts are not utilized as a reason to schedule user-level software for execution. Instead, a program typically polls a device or the operating system until it is determined that the event of interest has occurred. There is no way of determining how long between hardware interrupt detection and the user-level software's finding out. For this reason, time-slice operating systems are not generally described as being deterministic.

Traditional AT&T UNIX(tm) operating systems are not known as being deterministic in nature. To get around this, HP-UX/RT, for the HP 9000 Model 840, uses kernel pre-emption techniques. This means that the operating system has been enhanced to periodically check to see if interrupts have occurred, even in the middle of tasks. By doing this, observed response times have improved remarkably.

Beware of another computer industry definition of real-time. In the commercial computing environment, real-time refers to transaction-based systems. These systems are used in such applications as automated-tellers and airline ticketing. Make sure that when you talk to people about real-time applications that you both agree on the same working definition.

Single-tasking operating systems, i.e. those which run only one user program at a time, can be just as effective, or more so, for real-time response than multi-tasking systems. This is due to the simplicity of their operating system design.

VIEWING THE PROBLEM IN TERMS OF THE MEASUREMENT TASK

Many people have discovered the benefits of automating manual testing—more measurements in less time, increased efficiency, greater measurement and documentation consistency—all of which leads to higher productivity on the part of the individual responsible for testing.

Software automating the measurement process exhibit surprisingly similar design. In data acquisition programs, the task consists of doing some setup work, taking the basic measurement(s), doing some computation to yield the desired answer, presenting the answer, then perhaps, going back to do it all over again. (See Figure 1.) Process monitoring and control works similarly.

Data Acquisition

- [1] Setup
- [2] Make measurements
- [3] Do computation
- [4] Present results
- [5] Store results
- [6] Go back to [2]

Process Monitoring & Control

- [1] Setup
- [2] Make measurements
- [3] Make decision about
process
- [4] Make needed process
adjustments
- [5] Go back to [2]

FIG. 1: THE TASK AT HAND

In the above lists you will find typical actions in the order in which they are performed in many automated test and measurement applications. As examples, these lists provide a framework for discussing the issue of how to select an instrument controller.

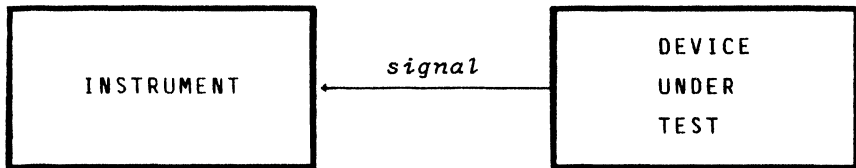


Figure 2A: THE MEASUREMENT -- Simplest Case

All automated processes must start with "the measurement." Some device or comparable source produces a signal to be measured by the instrument. In this simplest case, the signal is produced constantly.

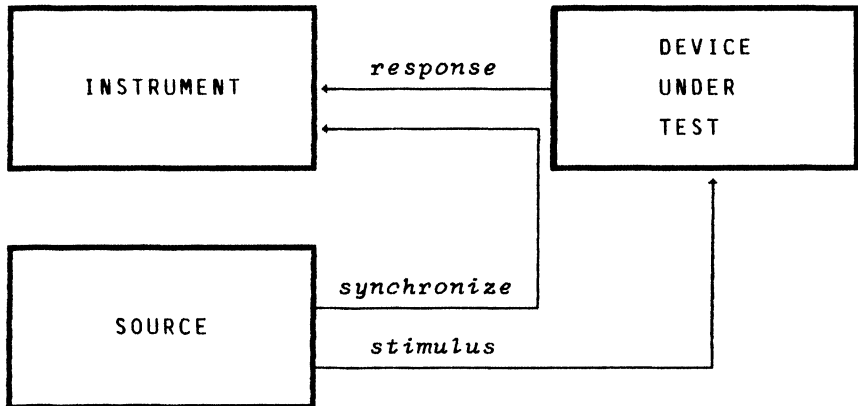


Figure 2B: THE MEASUREMENT -- Common Case

Most measurements are not made on devices that simply produce a signal. More commonly, there is a signal source which stimulates the device to be tested. The responding output signal from the device is then measured by the instrument. A synchronizing signal often triggers the instrument to make the measurement at the right time.

Let's analyze the data acquisition process in terms of the task requirements, to see how it might affect the desirable characteristics of an instrument controller.

The Measurement

First, there is the measurement. You must verify whether or not the instrument can take it. (See Figures 2A and 2B.) Usually, this is done manually.

Of concern at this stage in the process is how the measurement must be made, whether or not the data must be brought in immediately, how much data must be brought in per measurement and over what period of time. These things determine just how quickly a controller must respond to the instrument and what kind of minimum throughput it must be able to sustain over a given time frame.

For example, the instrument might be a digital voltmeter (DVM). In the given test, for each trigger of the DVM, it may take 200 readings. At 14 bytes per reading, this translates to only 2,800 bytes per measurement. If the next test must be run within 30 seconds of the previous, the controller should be able to input the data from the DVM at a rate of something better than 100 bytes per second, allowing for two or three seconds to re-arm the instrument for the next measurement. This application probably can be run by a low speed controller, such as an HP-71B.

Figure 3: THE MEASUREMENT

- Can the measurement be made with the desired equipment?
- How quickly must the measurement be made and brought into the computer?
- How often must the measurement be made?
- How many data points are there per measurement? (How many bytes or words of data must be brought in?)

Other applications may require quick turn-arounds (for example, respond with output within 10 milliseconds of an input) on very short bursts of data (say, less than 20 bytes). Another may require a sustained throughput of 800,000 bytes per second for five minutes, producing a total of 240 megabytes of data. Obviously, the former will require a controller with a response time of five to eight milliseconds, in order to output in time. The latter requires high-speed data transfer, plus a high speed unit in which to deposit the data, e.g. a rigid disc drive. Figure 3 lists the questions which should be asked about this stage of the data acquisition task. The answers to these questions will determine the I/O characteristics of the controller.

Instrument to Interface

After verifying the measurement, the physical communication link must be established between the instrument and the interface installed in the computer. Of primary importance is whether or not an interface can be found to match the instrument's interface—mechanically (connector to cable to connector), electrically (signal levels, timing), and functionally (handshaking, protocols). It's often relatively easy to find interfaces matching in the mechanical and electrical areas. If you're handy with a soldering iron and wire clippers, you might be able to fix any disparity between voltage and current, or connectors.

The functional aspects of the communication link are more difficult. The IEEE-488 interface standard goes into quite a bit of detail to ensure that the mechanical, electrical, and functional aspects of a conforming interface are fully compatible. The EIA Recommended Standard RS-232-C, on the other hand, does little more than recommend the mechanical and electrical interface characteristics—inevitably leading to considerable confusion among users, due to disparate implementations. Figure 4 helps to illustrate.

Figure 5 contains the checklist for the physical communication part of the data acquisition process. Since I've already discussed the points brought up by the first three questions, I'd like to concentrate on the rest.

Timing and speed is highly important to the physical communication layer. Handshake timings must be observed between the sending and receiving interfaces, if data is to be transferred without error and within the desired length

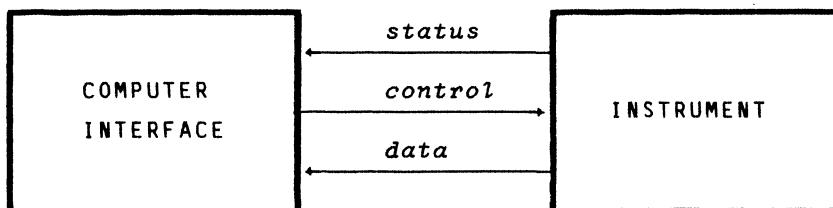


Figure 4: PHYSICAL COMMUNICATION

The interface for the computer provides the mechanism for matching characteristics with the instrument's interface. On the functional level, the interface provides the means to obtain status information from the instrument, to control the operation of the instrument, and to obtain measurement data from the instrument.

be transferred without error and within the desired length of time. For example, under certain circumstances with a two-wire handshake, the computer's interface may get ahead of the instrument's interface by changing the data lines before the instrument has finished processing the data. The three-wire handshake patented by Hewlett-Packard, on the other hand, guarantees no data loss due to the handshake timing. The three-wire handshake is utilized by IEEE-488.

Hardware buffering, either in the instrument, or on the computer's interface card can affect the timing/speed characteristics desired of the controller. If there is buffering in the instrument, multiple readings or measurements may be stored up until there is enough to send to the controller. This will loosen response time requirements for each discrete item to bring in from the instrument.

Buffering on the interface card will help to increase the time frame during which the interface driver must process incoming data (bring it into the operating system from off the card). Buffering can also help to reduce the number of times the driver must push data out onto the card for output, by allowing it to hand a number of bytes or words

to the interface, and let it pace the output to the receiving device. Without buffering, the driver would have to hand the interface one byte or word at a time.

Let's look at an analogy: It's Sunday morning, and you would like to make scrambled eggs for the family. You find that you need to go to the store for the eggs. You could go to the store and buy a carton containing one dozen eggs. You could also go to the store, buy one egg, bring it home, then repeat the process until you have enough eggs for breakfast. The first method is similar to buffering, the second, to handing an interface card one byte at a time.

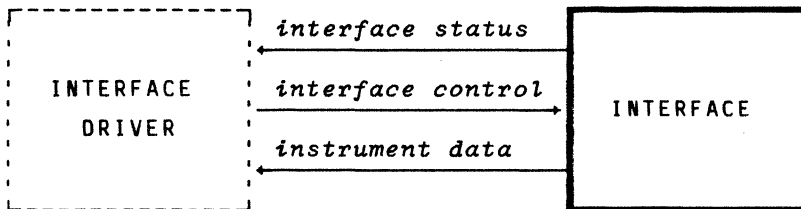
**Figure 5: PHYSICAL COMMUNICATION BETWEEN
INSTRUMENT AND COMPUTER INTERFACES**

- Is there an interface available and room for it in the controller package?
- Does the interface match the electrical characteristics of the instrument interface?
- Does the interface connector match the cable connector? (e.g. wiring, plugs)
- Is the correct cable/connector available?
- Can the interface match handshake timing with the instrument's interface?
- Can the interface produce the protocol expected by the instrument's interface?
- Does the interface interfere with anything else in the computer system? (Especially if you make your own interface, or bought a third party's interface card.)
- If there's no interface (e.g. you made your own device), is there enough information available about the computer to make one?
- Does the instrument have hardware buffering?
- Does the interface contain some amount of hardware buffering?
- How intelligent/complex is the instrument? Can it send a signal to the computer's interface regarding measurement completion or detected errors?

You may get the impression that hardware buffering is much more efficient than the one-at-a-time approach. Well, maybe. It all depends on your computer and its operating system. In a system with a quick response time, it may not matter which way the hardware is designed. Some systems, however, have more difficulty dealing with a large amount of data in per I/O transaction. So, it may switch to processing smaller packets of data, thereby slowing the process down anyway. In such a situation, it may have the same or better throughput in the one-at-a-time approach.

Interface to Driver

From the interface card to the driver, the measurement has entered the software domain (see Figure 6). To this end, it would be wise to make sure that the software for the hardware exists in the form of the driver. If so, then the driver must be checked to see if it has full control of the interface card. That would guarantee maximum flexibility and capability of the potential instrument controller.



**Figure 6: MOVING THE COMMUNICATIONS TASK FROM
HARDWARE TO SOFTWARE**

The interface card must have a piece of software to run it, so the operating system can perform I/O. In order to control the interface, you must be able to obtain status information describing the interface's current condition or state. All non-interface-related information from the instrument, is transferred to the driver as data. This includes instrument status and commands, and the measurement data. Instrument control is also sent typically as data through the interface.

Control of something begins with knowing what state it's in. The driver must be able to interrogate the card for current operational settings, error conditions, and so on. This way, errors and abnormal interface states can be detected and handled. Status information is typically stored in registers on the card. The information read back is interface-dependent in meaning. This makes sense because different interfaces have different operational characteristics. (You don't normally ask an RS-232-C interface if it is a system controller—a function associated with IEEE-488.)

Control of an interface card means the ability to change its operational characteristics. This is typically done by setting certain registers on the card which have specific meaning to that interface. Each type of interface, like IEEE-488 or RS-232-C, will have different register definitions: Some RS-232-C cards can be told to change transmit/receive bit rate and error detection mode (parity). In IEEE-488, it is necessary to control the state of the attention line to define the meaning of data lines.

If the interface driver doesn't provide full control, it's a good bet that you'll need to control a device some day that will require that control, and you'll be out of luck. So, go for driver flexibility.

You have now determined that the driver is available, etc. Now you must determine if the driver can keep up with the interface in a high-speed application, or, in a low-speed application, that the driver mustn't over-run the interface.

One note: Make sure that the driver is able to transfer all data to and from the interface. Many interface drivers will provide transaction termination detection and control. For example, an instrument may require all commands to be terminated with a certain byte sequence, say *carriage return*, followed by a *line feed* (ASCII decimal codes 13 and 10). These commands, routed through the driver as a data stream, may have the termination sequence appended to it by the driver, rather than forcing you, the programmer, to embed it in the data stream. On input, these data item separation bytes may be stripped, so you don't have to worry about them.

In another instrument, the *carriage return/line feed* sequence may happen to occur in the measurement data by coincidence. The driver may decide to remove those bytes

from the measurement information, or worse, stop inputting the data, having interpreted the bytes as a indicator to terminate the input. You'd like the driver to have the capability to switch from attaching meaning to the transmitted data to a transparent mode. This mode would force the driver to transfer the data "as is." Figure 7 lists the questions to be asked of the driver-interface interaction.

Figure 7: MOVING THE COMMUNICATION TASK
FROM HARDWARE TO SOFTWARE

- Does the driver exist?
- If I have to write my own, is there enough information and support to do this?
- Does the driver coexist peacefully with everything else? (Especially if I write my own or buy a third party driver.)
- Can the driver obtain all interface status?
- Can the driver transfer data to and from the interface without overrunning it or losing data?
- Can the driver transfer data transparently?

Driver to Operating System

Once data has reached the driver, it must be passed on to the operating system for later application program access.

Figure 8 shows how the driver communicates with the operating system. Timing continues to play a heavy part. It does no good to have a high-performance driver if the operating system drops the ball, and vice versa. Timing checks must be made on response time to the driver and throughput on buffered transfers (I/O with data being moved to and from blocks of specially-designated memory--buffers). This was touched on earlier.

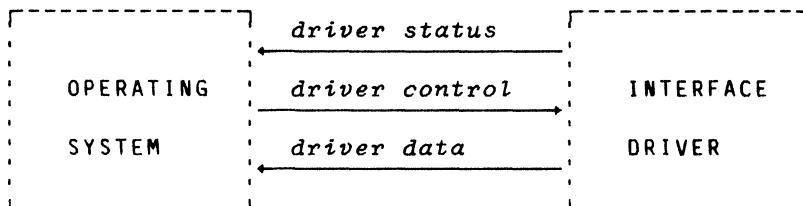


Figure 8: OPERATING SYSTEM AND INTERFACE DRIVER

The interface driver is used by the operating system to provide the control software for the interface hardware. The driver may or may not transform the data as it goes to and from the interface. Instrument commands pass through the driver as data. Control over the interface must be implemented as control commands to the driver. Status of the interface may be returned via the driver status or as data. Measurement data is passed as data from the driver.

Figure 9: OPERATING SYSTEM INTERACTION WITH THE INTERFACE DRIVER

- Does the operating system fully utilize the driver?
- Does the operating system provide application program access to the driver? What's the mechanism provided by the operating system for programmatic access to the driver?
- Can the operating system communicate with the driver quickly enough to accommodate the timing needs of the external device?
- Can the operating system keep up with the driver in a large, high-speed buffered transfer?
- Can the operating system respond to the driver quickly enough so as to not lose any data? Or lose precious time in the control loop? Or, in production, hold up the line?

Another consideration, as pointed out by Figure 9, is the operating system use of the driver. The interface driver may make full use of the interface, but if the operating system hasn't full control over the driver to make use of all it's capabilities, there could be trouble.

Operating System to Your Program

By now, it should be clear that your program must go all the way through the computer's operating system to talk to the instrument. (Figure 10 illustrates.) The operating system, being the glue that ties everything together for the computer, therefore, generates the longest list of concerns. The list may appear intimidating at first, but it's really just an extension of the previous levels' capabilities.

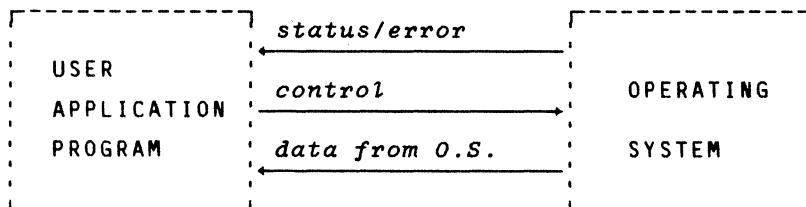


Figure 10: USER PROGRAM AND OPERATING SYSTEM

The application-level program must be able to request data transfers to and from external devices, such as instruments and terminals from the operating system. It should also be kept apprised of the results through error and other status messages.

The gist of the questions about the operating system have to do with the dilemma: "Is this thing going to work for me or against me?"

Areas which commonly cause problems at the operating system level include the following: lack of access to the interface driver; lack of high-level task support (e.g. no formatter or human interface device drivers); lack of

support for external event detection. In Figure 11, these have been organized into two categories: operating system intrinsic capabilities (Fig. 11A), and utilities (Fig. 11B).

Figure 11A: PROGRAM INTERACTION WITH THE OS:
Intrinsic Capabilities

- Does the operating system design define I/O as file system I/O? If so, is there any way around it?
- Does the operating system provide low-level user access to system resources, such as to interface and device drivers? If not, can the system be bypassed--with how much effort?
- Is there user-level control available over the I/O method, i.e. with or without DMA, byte-by-byte, buffered, transparent, etc. and how I/O transactions are terminated?
- Does the program have access to event detection information? If so, then in "real-time"? Can routines or tasks be executed upon event detection?
- Can the program transfer large blocks of data at a high-enough transfer rate?
- If resources aren't readily available to the user level, can any user-written software bypass the operating system? If so, is there enough documentation and support available?
- If the hardware configuration is changed, how much impact will this have on user software, i.e. how well is software insulated from driver and hardware differences?
- In a multi-tasking environment is there...
 - ✓ user-level control over task scheduling in the form of task priorities and the ability to lock a process into main memory and/or to the CPU?
 - ✓ interprocess communication, in order to be able to synchronize tasks and share data?

OS Intrinsic. The operating system may have been designed to consider interface and device drivers its exclusive domain. There may be indirect access to these by making certain I/O requests, yet the user has no low-level access. This is characteristic of a "closed" system.

A closed system may not be a problem if its designers provided some very fancy I/O routines for the user. For example, BASIC, as an operating system on the HP Series 80 and HP 9000 Series 200/300, is closed. However, it works very well by providing language extensions that have a high degree of control over the interface driver.

An additional problem with many general purpose operating systems is their insistence on considering file I/O to be the same as device I/O, that is, I/O to instruments. In these systems, it may be possible to bypass the file subsystem, if the system is open (has hooks for users to change or bypass the normal operating system and has enough user-available documentation and support to be successful).

A warning about attempting bypassing or changing an operating system: It isn't easy to force a system to do something for which it was not designed. It takes a lot of time to implement. Furthermore, it requires extensive testing to ensure that the changes have not affected the normal system operation. Thus, in the end, it may cost far more in engineering labor than it would have cost to purchase a system which was designed to perform the necessary I/O. The same goes for "rolling your own" operating system.

Events are another area of concern. When an interrupt occurs on an interface, it may have to be serviced immediately. If your program has no way of detecting that interrupt as an event, you're out of luck.

At the very least, the controller's operating system must provide the information that an event has occurred. A data acquisition program could continually ask the operating system: "Has the event occurred yet?" This method is known as polling, and works quite well in applications where other things don't have to be done during the waiting period. It also works okay in situations where other things should be done and timing isn't too critical.

Event-driven programming is easier to design and implement when the operating system can be instructed to execute specially designated code when the designated event has occurred. Combined with polling, it is among the most

desirable attributes of an instrument controller. It can be used in the situation where tasks other than waiting for the event to occur must be done.

Speed is an issue if response time is critical to the application. The time it takes the operating system to decide that the event has occurred, stop the currently executing code, save the current state, and initiate the event-servicing software must be less than the required response time.

Multi-tasking operating systems can take much longer at event response than single-tasking operating systems because of the problem of saving appropriate information and starting the event-servicing code. Depending on how much the operating system must save and do to start the next will significantly impact response time: Compare the HP 9000 Series 200/300 Pascal Workstation (single-tasking OS) vs. HP 1000 A900 (real-time, multi-tasking OS) response times in the Controller Survey section.

These attributes of event (interrupt) detection, event-driven programming, and "satisfactory" response time are what people involved with instrument control often mean when they refer to "real-time".

Of the items listed in the checklist, one other item should be noted. Many operating systems do not do a good job of insulating the software environment from changes in the hardware configuration of the system. For example, if a change in an instrument's address is made, or another one added, it may require taking the system down to change the operating system: In the case of HP-UX on the HP 9000 Model 840, the operating system must be re-compiled with a new device table so the drivers know what to do with the attached devices. In the case of some MS-DOS computers, changing the CRT can have profound affects on programs producing graphics because of the design of the supplied graphics libraries. Conclusion: If the controller is to be in an environment of constant hardware change, or potential change, it would be a good idea to look into this issue.

Finally, the issue of a multi-tasking environment should be discussed, since it adds complexity to the issue of instrument control.

The capability of having more than one program handled at the same time by the operating system can make the job of subdividing the tasks of measurement automation program

easier. Each task can be a different program, making the debugging a little more straight-forward. One program can be an event-handler. Another can just deal with the user. Yet another can compute the results of a measurement, then insert it into a data base.

It would be important to associate priorities with these different programs. The event-handler should have the utmost importance, at all times. If data starts bursting in from an unbuffered, non-handshaking device which transmits in bursts, at unpredictable times, such as a satellite, it must be dealt with immediately: The user can wait.

Many multi-tasking operating systems allow the user to set priorities on different programs. Essential in the satellite example, however, is the ability to also tell the operating system, that it should immediately suspend any other task, even if it's the operating system itself. In some general-purpose operating systems, such as AT&T's UNIX(tm), priorities may be set, but the operating system cannot be disturbed. The kernel of UNIX cannot be preempted. One of the things that was done to make HP-UX on the HP 9000 Model 840 respond more quickly was to provide for kernel preemption.

Another thing that was done was to provide a special kind of priority, real-time, that also does not degrade in importance. In many multi-tasking systems, including UNIX, priority will degrade over successive time-slices, so as to give other programs a "fair chance" at executing. In real-time situations, such a priority scheme will give the response time a "fair chance" of failing application requirements. Non-degrading priorities are a must.

When the event occurs, it is essential that it takes as little time as possible to find the event-handling program. The best place to find it would be in main memory. A desirable characteristic of the system would be the ability of the user to specify that the program is to be locked in memory. Otherwise, the operating system may swap it out to disc, while other programs have their turn at execution. The ability to lock a program in memory is known as "process locking."

When the instrument or device requires immediate response from its controller after sending data, it makes sense to also lock the program to the CPU itself. Let's see why: In most multi-tasking systems, if the program issues a request to input or output data, it is temporarily

suspended from execution, awaiting completion of the transaction. If the program must input data, evaluate it, then issue an immediate response, that lag time of being suspended, run, suspended, etc. could be disastrous. Imagine the consequences in a situation where people's lives depend on the controller's performance, such as on a factory floor where the computer may be the master of heavy mechanical equipment.

Assured that the real-time programs can run properly, how does it communicate with the other programs, to let them know that data is available? Various mechanisms have been developed for "interprocess communication." One of the oldest and easiest to implement is to set aside a special area of memory which has been defined as "shared." Programs depending on the actions of other programs can access and change this area. The programs must agree on the definition and rules for handling the shared memory, in order for it to work properly.

Newer methods of communicating between programs include signals and semaphores. This requires the operating system to have the ability to carry messages between programs. Data still may have to be shared, but task synchronization is made easier for the user to implement with semaphore support. Thus, instead of continually checking memory to see if the measurement data is available to process, a program could "wait" for a message to that effect.

Language and Utility Support. It may be all very good that the potential controller's operating system supports all of the capabilities mentioned above, plus direct driver access, but if the system supports no high-level languages which also provide access, what good is it? That's why the issue is mentioned in the checklist in Figure 11B.

It also makes sense, to look for support of as many other utilities that would help to shorten the project development cycle as possible. Much of the time spent writing instrument control software is in getting the data transformed into the format that the instrument needs to see it in on output. On input, more time is spent writing code to transform the measurement and instrument status information into a usable form--in other words, formatting! The computer should have some general-purpose utility to help out with this chore, or the high-level language of choice should.

Another area that takes a lot of time to implement, and the area that may make or break a software package, is the user

communication portion. The operating system must support the required human interface devices in a high-level fashion. This is especially true of complex ones, such as graphics devices.

Figure 11B: PROGRAM INTERACTION WITH THE OS:
Language & Utility Support

- Does the operating system support a high level language that allows full access to its resources?
- Once the data has been brought in or is ready to be sent to the device, are there utilities (such as a formatter provided by the program language) to help to transform the data into the desired format?
- Can I communicate with the user in a simple and satisfactory manner? That is, is there a high-level, flexible...
 - ✓ graphics library?
 - ✓ printer and CRT control?
 - ✓ human input device control?

Putting It Altogether: The Big Picture

It should be clear that looking at the problem from the measurement perspective helps to clarify what characteristics an instrument controller should have, in order to perform the job adequately. (See Figure 12.) There are, however, a few more areas of concern, involving the specific requirements of the measurement automation application and its implementation.

Application requirements. The type, size, complexity, and environment of the application must each be considered when making a controller selection. For instance, the larger and more complex the application is, the more likely it won't fit or run efficiently on a PC or a single-tasking system—and don't forget the amount of data to be acquired and manipulated. Furthermore, in a constantly changing application environment, such as test software development

in a university or in a company's R&D department, the existence of simple, quick and well-supported program development tools is necessary. (See Figure 13.)

The physical environment defines the environmental specifications for the potential instrument controller. Take the office environment: It's pretty forgiving on equipment. However, unless a piece of hardware is built for such places, the desert, glaciers, foundry floors, or moving vehicles can be disastrous. Consider, too, if the equipment has to be moved around a lot. Equipment, such as certain disc drives, can be moved, but must be specially handled during the move and recalibrated after the move.

Reliability is also important. If that satellite must be monitored 100% of the time, or if a worker's life can be endangered, you must make sure of the guaranteed uptime for the potential controller—or see if it supports fully redundant system configurations.

Does the measurement result have to be communicated to another computer system? If so, then networking, or data communications support is important. Simple file transfer mechanisms will work in this case. Some systems, say those in a CIM environment, also require the capability to take orders on what test suite to run next.

The complex world of networking is one where standards and compatibility are critical to the success of communication. This is not the place to discuss it at length. Just be advised that if this is a required part of the application, that you should be insistent upon standards compliance and full support of the necessary networking capabilities.

The number of people who have to use the system, the number of different kinds of tasks the system has to support, and when, is another issue. If two or more people need access to the computer at once, or if two or more completely separate tasks must be run simultaneously (e.g. payroll and measurement automation), then a multi-tasking system may be called for. I use the word "may" because a distributed system may accomplish the same objective.

Choice between the two depends on other factors, such as initial system cost versus cost of adding new stations later. A multi-tasking, multi-user system typically costs much more at initial purchase than a small network of single-user, single-tasking systems. However, adding a terminal to the multi-tasking system later will cost much less than adding another computer to the distributed

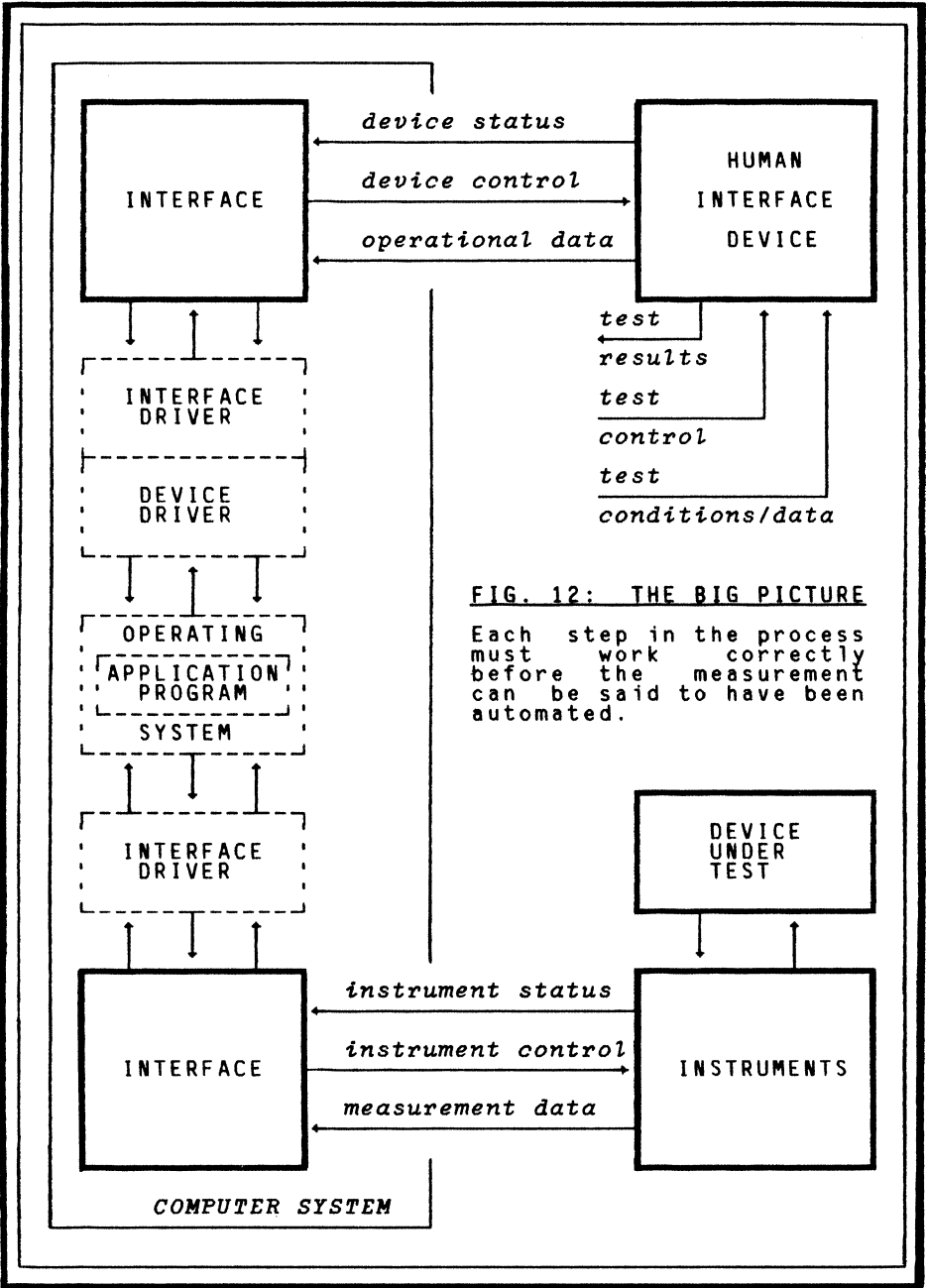


FIG. 12: THE BIG PICTURE

Each step in the process must work correctly before the measurement can be said to have been automated.

system. Going with the distributed system does allow you to start with one or two functional systems, then add one or two per year--a boon to the small yearly budget.

Figure 13: APPLICATION CONSIDERATIONS

- How often will the application software be modified, added to, or updated?
- Will software development and use of the system as a controller have to be at the same time?
- How big is the application software likely to be?
- Into what kind of physical environment will the system be placed?
- What is the projected number of users for the system?
- What is the application speed requirements?
- How much on-line and off-line storage is needed?
- How is the human interfacing to be done?
- What is the projected system use over time?

Another consideration is uptime. If you can't afford the whole system to be down at once, then the distributed system may be the better choice. On the other hand, if sharing information between users and tasks is of utmost importance, you may not be able to afford to have the network go down. The multi-tasking system is a better choice in that case. So, the bottom line is that it's all a matter of trade-offs. You pick the system based on your particular application's most important requirements.

Speed has been discussed a lot, yet it needs one final mention. Computation speed of the computer will affect the final, overall throughput of the application. This is because, as pointed out earlier in Figure 1, it's very rare for the computer to just take a measurement. Some manipulation of the data must be done in order to yield the results which, in turn, must be dealt with somehow: communicated to the user; sent to another computer;

stored for future analysis, etc. Thus, the time to manipulate the measurement data will add to the I/O time to determine application throughput. In the end, the last is the only figure of merit that counts. It answers the question, "Can I get the answer in time?"

Earlier, hardware issues were touched on briefly, in terms of overall system architecture, and availability of interface cards. There are other things which should be checked. These include RAM and peripheral availability and expandability.

Be sure to get a system that has enough memory to run the operating system and application software, plus space for data. Don't skimp on memory. It's easy to keep adding to application software until it fills available RAM and then some. In a system which cannot dynamically load new code segments, this could make the system rather unhappy. Attempting to set aside more memory locations than is available for data also could cause the system to stop. If that isn't the case, and most of the system RAM is being utilized, a true performance problem may occur. Waiting for new code segments to load, or parts of RAM to be released back to the system, contributes to performance degradation.

Another reason to make sure that the computer has RAM to spare is that new revisions of the operating system or application software are rarely smaller than the last: Those features that everyone asks for has to take its toll somewhere! Make sure that you have enough room to grow over time.

Once the memory configuration question is settled, look at what peripherals are available for the potential controller. Peripherals can be grouped into such categories as mass storage and human input/output devices. These should be examined separately.

The great majority of instrument controllers use some sort of mass storage device, if only to boot the operating system. These devices may be disc drives for flexible or rigid discs, tape drives, or even semiconductor memories, such as EPROM and bubble memory. This mass storage is used for operating systems, system software, utilities, application software, and data. In performing this function, it becomes a vital part of the system. It can affect the system throughput. It is also keeper of all that priceless information.

When deciding on mass storage peripherals, be sure to include methods or devices to make copies of important information. When calculating the minimum requisite storage capacity, be sure to include the operating system size and needs (it may need extra space for system functions), and program and data space. Be sure to leave plenty of room for growth.

By the way, be careful with disc drives in hazardous or extremely dusty environments. The drives are not as robust as the computers, due to their mechanical nature. Vibration and shock are a problem for most drives, especially those with hard discs. Dusty environments can leave coatings of particles on the mechanical parts, causing wear or corrosion. Use drives with sealed head and disc assemblies, such as Winchester drives. Better yet, go with something like bubble memory, or a networked system.

Human input/output devices can be viewed as two basic types: alphanumeric and graphic. Alphanumeric devices include printers, terminals, keyboards, custom keypads, etc. Graphic devices include plotters and special monitors. Some devices are capable of supporting both types of I/O. Such devices include mice and touchscreens (in alphanumeric mode, they control alphanumeric cursors; in graphic mode, they control graphic cursors), terminals, and printers.

Many applications require limited I/O with the system operator. Production applications are an ideal example. Often, there is no room to have a keyboard, which may also add a confusion factor to a production station. Therefore, touchscreens, custom keypads, and barcode readers, make excellent input devices. A single light, voice output, or a monitor for output, also works. If your application is in this sort of environment, or an extremely hazardous one, check to see if the system supports these sorts of I/O devices. Often, third parties provide them with standard interfaces, like RS-232-C.

Perhaps I should discuss why graphics should be utilized in a data acquisition system. The abundance of graphics output devices indicate the human penchant for processing information graphically. Pictures are easier to assimilate, can contain more information in less area, transcend language barriers, and convey the information more accurately. Real-time graphics allows the user to see the current state of the process in a process-monitoring and control application—at a glance. Post-measurement graphics can show the waveform results, not just a single

pass/fail value, which can be of great value in R&D and quality assurance areas.

When configuring a system, don't forget future expansion possibilities. If there is even the remotest chance that the system might change in some way, then be sure that the system is extendable. Some areas to check are: memory, CPU power, more I/O ports, and new peripheral support. Also make sure that adding capacity to the system does not mean you should have a black belt in hardware modification, and have minimal impact on existing software. Many systems are difficult to upgrade and force significant changes to application software, which increases cost of ownership significantly and may invalidate support contracts with the original vendor.

Implementation concerns. Many of the things that can go wrong during the implementation phase of a project seem to be forgotten in the process of selecting an instrument controller. Yet, they are crucial considerations. The problems that arise at this time can determine whether or not the system will be successful. See Figure 14 for a list of those questions which should be asked about implementation issues.

The first question on the list deals with the existence of user-oriented tools to implement the system. I define tools in this case to be all those things which will help to satisfactorily complete the project. As such, I have included hardware, such as logic analyzers, protocol analyzers, etc.

Software tools that help are things like debuggers, languages, and utilities like data base managers, graphics libraries, and so on.

Documentation must be available in the form of data sheets, manuals, application notes, bug reports, etc. (This documentation must also be decipherable by the one(s) who must implement the system. I would recommend that you ask for a copy of the manual to peruse before system purchase.)

Often, the documentation may be very well done, but training is required to synthesize the information it contains for a given application. Training will also shorten the implementation time by introducing the information necessary to make the system work in a shorter time than it would take to learn the same information by learning it alone.

Figure 14: IMPLEMENTATION CONCERNS

- What development tools are available in the form of hardware, software, training, people, and published documents?
- What kind of support is available for the hardware and software?
- How much will the system really cost over time?
- What kind of compatibility is needed?
- How much time is available to implement the system, once it arrives?
- Who will implement the system? How much experience does he/she/they have? Is tending the system the only job responsibility?

People can be considered tools in the sense that they can be considered resources of information. The vendor of the computer should have specialists on the computer or the application of data acquisition. Users groups can be invaluable. Furthermore, you shouldn't overlook the people in your own organization that might already know something of the potential computer.

In addition to the existence of tools, make sure that support is available, both for hardware and for software. By "support," I mean that there is a way to get information about the system after purchase, that you can get the hardware fixed, get software updates, and so on.

Support costs can also be a way to evaluate potential controllers. The usual figure of merit is what percentage of the initial investment is the support cost per year, in dollars, where initial investment is the total cost of hardware, software, labor (including fees for consultants and contractors), and development time (e.g., as long as the system is down, X% of revenue are lost per day). The lower the number for yearly support, the better. It can be used to compare various computers; for hardware support, it can be used as an indicator of product reliability.

Cost of ownership is related: It's how much the system will cost to keep up over time. It includes support costs, whether you buy a contract or not. Obviously, it doesn't

make sense to buy a computer with a projected high cost of ownership: It would indicate that it breaks down a lot.

Compatibility is of two types: hardware and software. Typically, you look for something that is hardware compatible when the system vendor doesn't provide the desired part (or at the right price), or you're upgrading your present system. If you're likely to be in this situation, check to make sure that the new part doesn't affect any other part of the system.

In general, hardware compatibility is complete to a given level. Software, however, is another matter. Compatibility of features tends to be haphazard. If you are considering a purchase of "compatible" software, ask for a detailed list of differences, or better yet, get a demonstration.

Be careful about why you're asking for compatibility. For instance, if you want to transport the software you're about to develop to some other system in the future or to preserve currently running software, be sure that you are targetting a machine that will be a good instrument controller for your application. Furthermore, there is no guarantee that what is the desirable operating system or language today will be so tomorrow. De facto standards come and go in popularity, but real-world problems are always here. The features of instrument controllers haven't even come close to developing and maintaining de facto standards. Thus, even if you sacrifice some performance and development time to implement on a "standard" system, as soon as you add the instrument control, the system is no longer "standard."

It is also important to look at the development time that you will have to finish the project. The longer you have, the more complex the system you can afford to purchase. The less time you have, the more you should insist on a system with proven quick development time: The implementer won't have time to learn all the features and to spend a lot of time trying out ideas.

It's also important to consider the one(s) who will have to implement the system, since this will impact development time. If the programmer has little or no knowledge of computing or of the selected system, then training time subtracts from the available implementation time. Furthermore, it must be determined if the group needing the system can afford to dedicate one or more people to tend the system and do development. Many users cannot dedicate

a programmer or system manager to the computer. In this case, it makes sense to get a simple system with an interactive environment which provides immediate feedback and the ability to test out ideas quickly.

CONCLUSION

There's not much more to say on the topic of selecting an instrument controller. If you use the checklists provided to guide you in choosing wisely, you will avoid the problems listed in the last figure. In addition, I've included a survey of Hewlett-Packard computers with potential as instrument controllers. In the survey you'll find such information as I/O speed, operating system capabilities, recommended user backgrounds, and a list of application requirements that the computer in question will fulfill.

Good luck on finding the right controller.

Figure 8: COMMON PROBLEMS

- I couldn't get the computer to control the interface to the instrument because...
 - ✓ I couldn't get the right interface.
 - ✓ the interface wouldn't work right.
 - ✓ I had to write the driver and I didn't know how or didn't have time or didn't care.
 - ✓ I couldn't control the interface driver satisfactorily.
 - ✓ the driver had bugs and the vendor would not fix it.
- In my application, timing was a problem. I...
 - ✓ couldn't get data into the computer from the instrument fast enough.
 - ✓ couldn't analyze the data fast enough.
 - ✓ had trouble with the interface: It was too fast or too slow for my instrument.
- I couldn't figure out how to make the computer do what I wanted it to do because...
 - ✓ the documentation didn't make sense: It wasn't written so that I could understand it.
 - ✓ there wasn't enough information in the documentation.
 - ✓ there was too much documentation and I didn't know where to start.
 - ✓ the system wasn't sophisticated enough and I got tired of fighting it.

SURVEY OF HEWLETT-PACKARD CONTROLLERS

Handheld Computers

Product Family: Series 40, Series 70

Processor: HP proprietary

Product Description: Small, lightweight, highly portable (fits in the hand), low cost (\$1000), battery-powered. Supports HP-IL as primary interface. Converters to HP-IB, RS-232-C, and GPIO (general-purpose, 16-bit parallel) available.

Software: Single-user, single-tasking operating systems in ROM. Series 40 features a proprietary RPN-style OS, while the Series 70 uses an extended semi-compiled (tokenized) BASIC which provides some event-driven programming capability; furthermore, with the HP-IL interface module, high-level formatting commands are provided.

Some instrument software is available, especially for the HP 3421A Data Acquisition/Control Unit from HP's Loveland Instrument Division.

Data communication/networking is available through user-written software.

Recommended Use: Very slow¹, very small and simple I/O applications as a controller for low-cost test equipment in such areas as production process monitoring or laboratory bench tops, and portable data acquisition. Best transfer speeds are achieved on the HP-IL interface (on the order of 4 K-bytes per second, no formatting). HP-IB transfer rates on the order of 2.7 K-bytes per second can be observed when passing data from HP-IL to HP-IB in Translator Mode.

Portable Computers

Product Family: Series 80

Processor: HP proprietary

Product Description: Small, portable (generally fits under an arm, relatively low cost (less than \$5,000), extended BASIC language as the operating system. Supports standard instrument interfaces: HP-IB, RS-232-C, BCD (Binary Coded Decimal), GPIO, and HP-IL.

Software: Single-user, single-tasking operating system in ROM. This version of BASIC comes as an operating system and semi-compiled language. A series of optional ROMs can add language functionality and system capability. I/O support through the I/O ROM is quite good at both high and low levels. A high degree of driver and formatting control is provided, as well as event-driven programming. (This BASIC provides the basis for HP Technical BASIC now available as an interpreter on HP-UX.)

Many HP instrument divisions have software for the Series 80—especially the HP-85A.

Data communications/networking is available through terminal emulators and user-written software. Alternative operating systems are available for the HP-86 and HP-87: UCSD p-System² and CP/M(r).

Recommended Use: Slow¹, small, simple I/O applications as a controller for low-cost test equipment in such areas as production process monitoring or laboratory bench tops, and portable data acquisition. (HP-IB transfer rates may be observed at up to 26.2 K-bytes/sec. using fast handshake, unformatted transfers, and on the order of 19 K-bytes per second with GPIO.) Good for non-computer literate users or for those who have little time to spend programming. Use also in applications with short project implementation times.

Product Family: HP 9807 Integral Personal Computer (IPC)

Processor: MC68000

Product Description: Transportable (fits under an airplane seat), integrated (keyboard, flat-panel display, Thinkjet printer, double-sided 3½" floppy disc drive in one unit), low cost (starts at \$5,000), HP-UX-based system. Supports standard instrument interfaces: HP-IL, HP-IB, RS-232-C,

BCD, GPIO.

Software: The ROM-based HP-UX is AT&T UNIX(tm) compatible. It is also single-user and multi-tasking. HP Technical BASIC (upwards-compatible with Series 80 BASIC, including I/O support, and event-driven programming) is available in ROM or on floppy. Other languages available include C, Pascal, and Fortran. The HP-UX standard device I/O library and real-time extensions are supported. The user-interface to the system is through the Personal Application Manager (PAM) and windows, with each running program typically assigned to one window (including PAM).

Many Series 80 software packages have been ported and several of the technical office automation programs for HP-UX have been qualified.

Data communications/networking is available through *uucp*, terminal emulation, and user-written software.

Recommended Use: Use in slow to medium speed¹ applications requiring an industry standard multi-tasking operating system, or as an upgrade path for those using the Series 80 and could use a friendly introduction to HP-UX.

MS-DOS Computers

Product Family: HP-150, Vectra PC

Processor: Intel 8088 in HP-150, Intel 80286 in Vectra

Product Description: IBM PC compatible, desktop system, low cost (less than \$4,000). Instrument interfaces available include HP-IB, and the PC-Instruments bus for HP New Jersey Division personal instruments.

Software: Single-tasking, single-user industry standard system: MS-DOS³. HP-supported MS-DOS HP-IB Command Library, PC-Instruments, and ASYST⁴ analysis software provide low to mid-level driver control.

Data communications and networking capability includes terminal emulation, and HP LAN.

Recommended Use: Use in applications which need slow to medium¹ speed buffered transfers and slow response time, plus complete compatibility with IBM PCs or PC-compatibles.

Control of low cost instrumentation works well.

Language Workstation Computers

Product Family: HP 9000 Series 200/300

Processor: MC68000, MC68010, MC68020 (processor and clock rate depend on model)

Product Description: Desktop or rackmount (19" or taboret) versions, medium cost (\$6,000-\$30,000). Choice of operating system (BASIC, Pascal, HP-UX from HP; HPL, Forth from third party software suppliers). Standard instrument interfaces supported (depending on operating system): HP-IB, RS-232-C, GPIO, BCD, VME-bus, RS-422/423 and RS-449. A breadboard card is available. EPROM and bubble memory are available as mass storage alternatives. Internals documentation is available for the Pascal operating system.

Software: The language-based operating systems BASIC (often referred to as "Rocky Mountain BASIC" or RMB, for short), Pascal, and HPL are single-user, single-tasking systems. Forth is single-user, multi-tasking. A Pascal language compiler and an assembler are provided in the Pascal system. A Fortran/77 compiler is available for the Pascal system through a third party. The language systems provide both high and low level I/O support, with RMB being the most complete. RMB and HPL are semi-compiled languages and support formatting, event-driven programming, and background I/O transfers.

A lot of instrument-specific software is available, along with some technical office automation tools.

Much of the technical office automation software introduced recently have been designed to run under a low-cost HP-UX kernel, referred to as AXE. One of the things that runs under AXE is the MS-DOS co-processor, which allows the running of IBM PC/AT(tm) software.

Data communications/networking is available through terminal emulation and user-written software. Distributed processing available through a shared resource management system (SRM), supporting other HP 9000 Series 200/300 computers.

Recommended Use: Use in medium to very high speed¹ I/O

applications. (Best transfers rates can be achieved with DMA on the GPIO interface--on the order of 700 to 800 K-words/sec. or K-bytes/sec., depending on transfer mode. HP-IB transfer rates with DMA are on the order of 300 K-bytes/sec. Interrupt response times are on the order of 10 μ sec. in the Pascal operating system, while event response times in BASIC can be on the order of 2 to 8 msec.⁵) Box versions are suitable for hazardous environments. Good for non-computer literate or for those who can afford to spend little time programming. Use also in applications with short project implementation times.

HP-UX Computers

Product Family: HP 9000 Series 200/300/500/800

Processor: MC68000 for Series 200/300; HP proprietary for Series 500 and Series 800 (RISC-based)

Product Description: Desktop, floor, or rackmount (19" or taboret) versions; medium high to very high cost (\$15,000 to over \$150,000). Series 500 can be multi-processor (up to three CPU's in one system). Computationally, the Model 840 is fastest, being able to perform on the order of 1,927,000 double-precision Whetstones/sec. The instrument interfaces, GPIO and HP-IB, are supported. RS-232-C is supported through the standard terminal handlers.

Software: HP-UX is AT&T System V UNIX(tm) compatible. It supports some Berkeley enhancements, where not in conflict with System V. Languages available include C, Fortran/77, Pascal, LISP. Real-time extensions have been added, especially to the Model 840, in the form of kernel preemption capability and real-time priorities. Kernel preemption, when invoked, gives the Model 840 an average process dispatch time (in response to an interrupt) on the order of 1.5 msec. and a maximum observed one of 11.77 msec. A device independent I/O library, DIL, is a standard feature, providing low level I/O and buffered transfer capabilities. On the Model 840, a skeleton driver will be made available; source code also will be available to those with a source code license.

A growing number of technical office automation, data base management systems (including HP SQL), computer-aided design packages are available. Also available is a Common LISP environment for artificial intelligence tool

designers.

Data communications and networking available include uuop and HP's local area network. Access is also available through file transfer utilities to HP's SRM (except Models 207 and 840).

Recommended Use: Slow to high speed¹ I/O applications. If also computation-intensive, examine the Model 840. Recommended series and model depends on required transfer rates and response times. Good in complex applications requiring a multi-tasking, industry-standard operating system. Must have dedicated system administrator and programmers understanding multi-tasking environments.

HP 1000 Minicomputers

Product Family: E/F Series, A Series

Processor: HP proprietary

Product Description: Rack mount or floor models, medium-high cost (\$20,000 and up). Standard interfaces: HP-IB, parallel, RS-232-C, A/D, D/A, breadboard.

Software: Multi-user, multi-tasking, real-time operating system (RTE). RTE on the A900 yields a mean process dispatch time (in response to an interrupt) on the order of 710 μ sec., and a maximum observed one of 5.99 msec. Languages supported include Fortran/77 with military extensions, Pascal, C (from a third party), BASIC (both interpreted and compiled), macro-assembler, and micro-programming. User-written drivers are supported.

Instrument-specific, data base management, and general purpose technical application software is available.

Data communications/networking capabilities include distributed systems (DS/1000), X.25, HP's LAN, and user-written software.

Recommended Use: Medium to very high speed¹ I/O, complex and/or high speed computation-intensive applications. A dedicated system manager is required, along with programmers who understand multi-tasking applications. Frequently used in hazardous environments.

¹ A rough guide to I/O speeds:

Very slow: 0 to 5,000 bytes per second.
Slow: 5,000 to 15,000 bytes per second.
Medium: 15,000 to 50,000 bytes per second.
High: 50,000 to 100,000 bytes per second.
Very high: Greater than 100,000 bytes per second.

² UCSD p-System are trademarks of the Regents of the University of California.

³ MS(tm)-DOS is a U.S. trademark of Microsoft, Inc.

⁴ ASYST Scientific Software is a U.S. trademark of Macmillan Software Company.

⁵ Semi-compiled languages, as implemented by HP on the HP 9000 family of computers, log interrupts as they occur, but service them only upon completion of the current line of user's code. Thus, if the user's program has a PAUSE to wait for the operator to press the CONTINUE key, the theoretical response time could be infinite.

A GLOSSARY OF TERMS

This glossary is intended to be an informal list of common jargon used in the measurement automation world. It is by no means complete, nor has it been checked against any other industry-published dictionary.

batch. An operating system that runs queued up programs. The earliest method of running multiple programs, each program gets full utilization of the computer. Still used in commercial computing environments due to its efficiency in running very large accounting packages. [See also **timesharing**, **interrupt-driven system**.]

cache. High speed memory associated with the CPU. The idea is to reduce fetch times associated with executing instructions. It is generally of two types: instruction and data. [See also **CPU**, **instruction cache** and **data cache**.]

CIM. Abbreviation for computer-integrated manufacturing. [See **computer-integrated manufacturing**.]

compatible, compatibility. When some aspect of a computer's hardware or software behaves the same way as another. A computer may be compatible with another to the extent that it supports the most commonly used features of the other machine. The degree to which a computer or operating system is compatible is determined by the amount of change that must be made upon an item which was developed first on one machine, then transported to the one in question.

computer-integrated manufacturing. The process of automating the manufacturing process with computers and instrumentation, then linking them altogether with higher level controllers. The goal is to be able to integrate the whole manufacturing entity.

context switch. The process of suspending one process to

execute another. This typically requires three actions in most multi-tasking systems: (1) Process the event which is causing the context switch; (2) Suspend the currently executing process; (3) Begin executing the highest priority process. [See also **context switch time**.]

context switch time. The amount of time it takes to suspend the currently executing process, then begin executing the highest priority runnable process.

controller. A device that controls one or more other devices. Specifically, a computer which controls the operation of other devices. [See also **data acquisition computer**, **instrument controller**.]

cost of ownership. The cost of maintaining the system over time for both hardware, software, and support.

CPU. An abbreviation for **central processing unit**. That part of a computer which performs instructions.

data acquisition. The process of collecting data from the physical world, using electronic transducers and measurement devices (instruments).

data acquisition computer. A computer that acts as a controller to perform data acquisition. Such a computer requires special capabilities, including real-time response and a high degree of control over the I/O system. [See also **controller**, **instrument controller**.]

data cache. Cache reserved only for data. Its usefulness is dependent upon the likelihood that, as data is fetched from memory, it will be used over and over again, thus reducing the amount of time to process this data. [See also **cache**, **instruction cache**.]

data communications. The exchange of data between computers, or between computers and terminals. The communication is assumed to be via RS-232-C and/or over telephone lines. [See also **networking**.]

device driver. A driver which controls a device, such as a plotter or a disc drive. It often uses or incorporates an interface driver. [See also **driver**, **interface driver**.]

direct memory access, DMA. A method of performing I/O

operations as quickly as possible, with hardware assistance. There is more than one implementation scheme. One is to steal time away from the CPU to bring data in, on a cycle-stealing basis. In other words, when the CPU would normally be waiting for memory to return an instruction, that clock cycle is "stolen" to perform the I/O operation.

Another method, supported by the Motorola MC68000 family of microprocessors is to use an auxilliary processor, called a DMA controller, which specializes in moving data to and from RAM as quickly as possible. It is able to accomplish this by shutting down the CPU while the I/O operation is taking place. When complete, the DMA controller will release system control back to the CPU. [See also CPU, I/O, microprocessor, RAM.]

distributed system. A method of dividing up the workload and distributing computers located over a given area to increase overall computational power and capacity. The computers are linked together in some sort of network to coordinate the work and to share data. [See also network.]

DMA. Abbreviation for direct memory access. [See direct memory access.]

driver. A program associated with an operating system that controls some sort of hardware, such as interface cards or peripherals. There are two basic kinds of drivers: interface drivers and device drivers.

end user. A person who uses the computer in the manner for which it was intended. Usually, the end user knows enough about the computer to run specific application software.

event. The conceptual interpretation of an interrupt by the operating system of a controller.

friendly, user-friendly. A given task to be performed on a computer which can be accomplished with an acceptable level of frustration.

general purpose computer. A computer designed with no special application in mind, except general computation tasks. It may use either a batch or time-sharing operating system.

handshake. That sequence of signal changes between interfaces which allows the orderly exchange of information between them.

hardware floating point. Hardware designed to assist the CPU with floating point (real number) computation.

human input device. A device designed to let a user input information into the computer system. Input devices may deal with alphanumeric or graphical information. Examples include: keyboard, mouse, touchscreen, barcode reader, light pen, arrow keys.

HP-IB. Hewlett-Packard's implementation of IEEE-488. [See IEEE-488.]

IEEE-488-1978, IEEE-488. A popular interfacing standard designed to connect multiple intelligent instruments to one port in a bench-top environment. It is also known as HP-IB, HP-1B, GP-IB, the Plus Bus, the ASCII Bus, the IEEE bus. The latest revision was issued in 1978 by the Institute of Electrical and Electronics Engineers, Inc. (IEEE). It has also been approved by the American National Standards Institute (ANSI), hence the full name: **ANSI/IEEE Std 488-1978 IEEE Standard Digital Interface for Programmable Instrumentation.** [See also interfacing, HP-IB.]

instruction cache. Cache reserved only for instructions. Instructions are placed in the cache memory as they are fetched from main memory and executed. The usefulness of instruction caching is based on the fact that programmers tend to program in short loops or request relatively short branches. So, soon, the CPU should be executing instructions out of cache instead of lower-speed main memory. [See also cache, data cache.]

instrument. Electronic device that measures physical phenomena with various sensing devices, such as thermocouples. [See also data acquisition.]

instrument controller. A computer that controls instruments. [See also data acquisition controller.]

interface card. A printed circuit board providing the electrical, mechanical, and functional conversion between the host and another device located outside the host. This is accomplished by translating between the internal bus of the computer and the external bus

(cable). In a computer, it is typically installed into an I/O port. Intelligent instruments have interface cards built in. It is sometimes referred to simply as an interface.

interface driver. A driver designed to control a particular interface card.

interfacing. The process of connecting a device to a computer and establishing communication between the two.

interrupt. An asynchronous request for attention from a device connected to a controller. This request for attention is implemented in hardware.

interrupt-driven system. A multi-tasking operating system which allows interrupt-based events to determine the next task to be performed. The system may be exclusively interrupt-driven or use time-slices until an event occurs. An example of the latter approach is RTE (Real-Time Executive), the operating system provided for HP 1000 mini-computers.

interrupt response time. The time it takes for the computer to detect that an interrupt has occurred, it recognizes the interrupt as a permissible event, then starts to do something about it. What that something is depends on the system. Sometimes it's the time it takes to enter the interface driver. Other times it's the time it takes to enter the user's designated event-handling code. [See also interrupt.]

I/O. Abbreviation for input/output. [See input/output].

I/O subsystem. That part of an operating system which provides access to the interfaces and drivers.

I/O port. The physical connection point in the computer for an interface card.

input/output. (1) The process of moving data into the computer (input) and moving data out of the computer (output) via the I/O subsystem. (2) Storing (output) and retrieving data (input) from files on external mass storage. This definition is the more common one on general purpose computers.

LAN. Abbreviation for local area network. [See local area network].

local area network. A network restricted in distance between computer systems, so as to increase possible data transfer rates. An increasingly popular network design, IEEE-802.3 is a common basis for LANs.

mass storage device, mass storage. A device with the capability to store and retrieve data for the computer. Examples include hard and floppy disc drives and tape drives.

measurement automation. The process of making physical measurements an automated task, involving a controller and one or more instruments. The measurements taken may or may not be sensor-based. A more general term than **data acquisition**.

microprocessor. A processor that has been implemented on a micro chip. Often abbreviated as μP .

modem. Stands for **MOdulator-DEModulator**. In RS-232-C, it is a device which allows the conversion from the RS-232-C interface signals to a single signal which can be transported through the standard telephone system. Modems come in pairs: One is to go from RS-232-C to the telephone line; The other is to go from the telephone line to RS-232-C. This allows for a way around the limitation of 50 feet of wire between devices, as recommended by RS-232-C. [See also RS-232-C.]
RS-232-C

multi-tasking. An operating system characteristic: More than one user program can be executed at the same time. An operating system may be pseudo multi-tasking by allowing certain program requests to operate in parallel with the rest of the program execution. An example of this is starting an I/O operation, then going on to the instruction without waiting for completion of the I/O. [See also **time-slice** and **interrupt-driven**.]

network. A number of computers connected together with a data communications scheme so as to share information and tasks between them. It is usually based around RS-232-C. Using modems, there is no limit to the physical distance between systems (nodes). [See also **local area network, RS-232-C**.]

operating system. The software which provides access to

the resources of the computer. As such, it can run other software. Common abbreviations include OS and Op. Sys.

OS. Abbreviation for operating system. [See operating system.]

networking. The process of connecting computers together and establishing effective communication between them.

output device. A physical device used to output data from the computer, often in human-viewable form.

PC. Abbreviation for personal computer. Also sometimes used as an abbreviation for the IBM PC. [See personal computer.]

peripheral. A device which can be attached to a computer to perform at least one of a variety of functions along the lines of external storage, data input, and data output. [See also mass storage, human input device, output device.]

personal computer. A physically small computer that is intended for the use of only one person at a time.

priority. In a multi-tasking operating system, a value which can be assigned to a program or event and used by the operating system to decide which task is next.

polling. The act of requesting information from a device, such as an instrument or interface card. Polling can be done once or repeatedly.

preemption. The ability of the operating system to be briefly suspended from it's normal processing sequence to act on external events. This is done by having the operating system periodically check for these events, when it is safe to do so, meaning that the system is in a known, and keepable state.

preemption latency. The period of time the CPU is used by a system process before voluntarily giving it up to an event-handling process of higher priority.

process dispatch time. The time from when an interrupt occurs, to when the event-handling process starts running. In UNIX(tm) systems, it includes the interrupt response time, kernel preemption time, and context switch time. [See also interrupt response

time, preemption latency, context switch time.]

process monitoring and control. Found in manufacturing environments or other places where a computer automates a process: The job of watching over a process and making modification where and when necessary.

RAM. An abbreviation for random access memory -- electronic storage inside the computer, used for the operating system, programs, and data storage. Also known as memory.

real-time. (1) In scientific and engineering computing applications, an operating system implementation that allows as fast a computer response is as necessary to respond to physical stimuli (interrupts). Required response times vary from application to application. Thus, it is a relative, rather than absolute value. (2) In commercial applications, an operating system implementation oriented towards on-line transaction processing, such as airline ticketing. Response time is based on human response-time.

real-time computer. A computer with a real-time operating system. Implicit in the design is a special hardware architecture to improve I/O throughput and interrupt response.

real-time data acquisition. The process of performing data acquisition in real-time. This suggests a need to respond very quickly to interrupts and to sustain high rates of throughput.

real-time system. (1) A real-time operating system. In order to be real-time, an operating system must be able to respond to events in a real-time fashion. It implies a high level of user control over the I/O subsystem and some degree of multi-tasking capability. (2) A real-time computer system.

response time. In the most general sense, the time it takes the operating system to detect an interrupt and to do something about it. Some define it as the time it takes to detect an interrupt and enter the interface driver code. Another definition is the time it takes to detect an interrupt and enter the application program's event-handling code. There is no actual or de-facto industry standard definition for this term. [See also process dispatch time.]

responsiveness. An operating system characteristic - the kind of response time to expect for an interrupt.

RS-232-C. A recommended standard published in August 1969 by the Electronic Industries Association (EIA). It defines an "Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange." It details the electrical, operational, and mechanical aspects of the interface. Typical data terminal equipment (DTE) include computers, printers, and terminals. Data communications equipment (DCE) refers to modems. [See also modem, data communications, networking.]

semi-compiled. A method of translating user-written programs into a series of symbols, or tokens, whose meaning will be interpreted at run time. Semi-compiled languages have the advantages of interpreted ones, while running more efficiently. Advantages include an interactive programming environment and smaller programs. [See also tokenized.]

single-tasking. An operating system characteristic: It can execute just one user program at a time.

SPU. An abbreviation for system processing unit. It includes a CPU and supporting hardware, not directly relating to executing instructions, for example virtual memory and cache. [See also CPU, virtual memory, cache.]

support. (1) A service or collection of services provided by product vendors to help the buyer to use and maintain the product. (2) Tools designed to help the user to implement his application. Tools include software (such as debuggers and other utilities), and documentation (such as manuals, and application notes). (3) Is able to run particular software.

task. A logically related collection of procedural instructions executed by the system, e.g. a program.

throughput. In reference to I/O, a figure of merit used to measure I/O performance. It is calculated by examining how quickly a computer can input data from some source and deposit it in some destination, perhaps in a new form. In other words, it is the number of data items per second transferred from the input to the output. Often, data items are measured

as bytes. It is quite useful as a yardstick of performance in measurement automation applications characterized by the need to gather large bursts of data in a short amount of time.

time-sharing system. An operating system using time-slice architecture. An example is HP-UX, Hewlett-Packard's implementation of Bell System's UNIX (tm) operating system.

time-slice. In a multi-tasking operating system, a period of time that a particular task is allowed to execute in. If a task requests I/O, it may be suspended immediately (before the end of the time-slice) until the I/O operation is finished. Through a series of time slices, the program will run to completion.

time-slice architecture. An operating system design that uses the time-slice as the only method of running multiple programs at the same time. Programs include operating system and user programs.

tokenized. The state of a program, statement, or command after having been translated from a logical sequence of characters into a series of symbols or tokens, which will be interpreted and acted upon later.

turnkey software. Application software that does everything desired and requires little user knowledge of computing to run. It is typically not modifiable by the buyer.

virtual memory. The ability of an operating system, or the SPU and operating system, to manage a user's program space or data space as if it were using physical memory, when in fact it may be located out on disc. This frees the programmer from having to worry about the size of his program or amount of data in memory, since the operating system will bring in the pieces of program and/or data on demand.

LIST OF REFERENCES

Institute of Electrical and Electronics Engineers, Inc.
ANSI/IEEE Standard 488-1978: IEEE Standard Digital
Interface for Programmable Instrumentation. New York:
Institute of Electrical and Electronics Engineers, Inc.,
1978.

Electronic Industries Association. EIA Standard RS-232-C:
Interface Between Data Terminal Equipment and Data
Communication Equipment Employing Serial Binary Data
Interchange. Washington, D.C.: Electronic Industries
Association, 1969.

Hewlett-Packard Company, Inc. Student Guide: HP-IB
Instrument Control Using HP Series 200 BASIC, course
number 50011B. Mountain View: Hewlett-Packard Company,
Inc., 1985.

ACKNOWLEDGEMENTS

I'd like to thank the following people for their help in the original research for this paper: Albert Alcorn, Data Acquisition Systems Engineer, HP Neely Pleasanton; Jim Gendreau, Communications Manager, HP Data Systems Division; John Jensen, Dynamic Signal Analysis Systems Engineer, HP Neely Santa Clara; Don Richmond, District Systems Engineer Manager; Kent Simcoe, Data Communications Systems Engineer, HP Neely Pleasanton; Dave Whitton, Microwave/RF Measurements Systems Engineer, HP Neely Palo Alto.

Two people who endured constant iterations of this paper (and who cried out, "Oh, no! Not again!") in order to provide technical expertise and general support deserve special mention: Nigel Clunes, HP 1000 Systems Engineer, HP Australia and Alan Tibbetts, consultant, Telos Consulting Services.

Dave Myers was a sales representative who wanted a white paper on how to choose instrument controllers, so he could help his customers make a good decision. That was in February 1985. I originally promised to have it done by that April. It was actually finished in November. Dave was promoted to Sales District Manager that summer, however. He still encouraged me to finish it: Thanks, Dave.

Software Management Strategies

**by: Miller, William
Clunes, Nigel**

We regret that this paper
was not received for
inclusion in these proceedings.

THE FIRST HP PRECISION ARCHITECTURE IMPLEMENTATION

David Fotland
Hewlett-Packard Co.
11000 Wolfe rd
Cupertino Ca 95129

Introduction

The HP Precision Architecture is based on reduced instruction set (RISC) principles but goes beyond RISC to add features in the areas of floating point support, virtual memory management, and I/O. Simplified instruction sets allow processors to be less expensive and faster when executing integer based programs. Provision for coprocessors allows high speed implementation of floating point, hardware multiply, and other similar functions. Paged virtual memory allows efficient execution of very large programs. A new high performance I/O system is needed to keep the processor busy.

The HP9000 model 840 is the first technical version of the HP Precision Architecture computer family. It runs the HP-UX operating system and is 2 to 3 times faster than an HP1000/A900 or an HP9000/550. The HP3000 model 930 is the first commercial version of the HP Precision Architecture. It runs the MPE operating system and is about 2 times faster than an HP3000 series 68. The model HP9000 model 840 and the HP9000 model 840 are very similar internally. This paper will concentrate on the HP9000 model 840. The model 840 is built from off the shelf TTL MSI, PALS, and RAMs. It has a three stage instruction pipeline allowing it to execute about 4.5 million instructions per second (MIPS). A 4096 entry translation lookaside buffer (TLB) and a 128 Kbyte cache system allow address translation and memory accesses to keep up with the processor. Main memory is implemented with 256 Kbit dynamic RAMs and can be up to 24 Mbytes. The connection between the processor, memory controllers, and I/O channels is a new synchronous bus with 20 Mbytes/second bandwidth. A CIO bus is used for I/O cards. This is the same I/O bus as on the 9000/550.

The Processor

The model 840 processor is implemented on three boards of TTL logic. Each board has about 150 ICs. These boards are the Instruction Unit (IU), the Execution Unit (EU), and the Register Files (RF). The processor has a 125ns cycle time and can fetch a new instruction each cycle. This implies a maximum performance of 8 MIPS. Precision Architecture instructions are all 32 bits long and have very similar formats. No instruction has more than two source registers and the source register

numbers are in the same place in every instruction. This makes it possible to fetch and decode an instruction every clock cycle.

The IU contains the program counter (PC) and the logic to update it. It executes branch instructions and has the PC incrementer and the PC offset queue. It also handles traps and interrupts. The RF contains the 31 general purpose registers, copies of 16 of the control registers, the external interrupt register and several other control registers. The EU contains the main ALU and the barrel shifter used to execute instructions as well as the condition generation logic.

The three cycles of the pipeline are called fetch, execute, and load/store. During the fetch cycle the IU sends the program counter (PC) to the instruction cache and TLB. The returned instruction comes to all the boards where it is decoded. The RF board reads the source registers and the IU increments the PC. During the execute cycle the EU calculates the result for this instruction and sends it to the RF. If the instruction is a load or store, the EU also calculates the address and sends it to the data cache and TLB. The EU also calculates the condition to be used in a conditional branch or trap on condition. The condition goes back to the IU. During the load/store cycle the result is written back to the register files on the RF board. If the instruction is a load or store the actual reading or writing of the data cache takes place during this cycle. At any one time one instruction is in fetch phase, the preceding instruction is in execute phase and the one before that is in load/store phase.

Floating Point

Pure RISC architectures generally have poor floating point performance so the Precision Architecture allows a coprocessor for floating point. The floating point coprocessor has its own 12 floating point registers so as not to interfere with the main processor. The main processor executes coprocessor load and store instructions to update these registers. Coprocessor operation instructions (such as floating point add FADD) take one cycle in the main processor. They then execute on the coprocessor in parallel with instructions on the main processor. This overlapping allows higher performance. The floating point coprocessor in the model 840 is based on an HP proprietary 3 chip set implemented in the NMOS III process. This is the same chip set used in the HP9000 model 550. In addition there is a register file and micromachine built from standard TTL. The micromachine handles self test, instruction sequencing, and exceptions. The model 840 is about 2 times faster than the HP1000 A900 and about 3 times faster than the HP9000 series 550 when executing floating point intensive programs such as Spice, Whetstone, or Linpack.

Cache and TLB

The Precision Architecture has an unusual virtual memory system since the virtual addresses are 48 or 64 bits. The model 840 uses 48 bit virtual addresses. There is a single virtual address space for the entire system rather than one per process. The mapping between virtual and physical addresses is one to one. This virtual address scheme has several advantages. Since the mapping is one to one it is possible to index into the caches with the virtual address. This speeds up the machine since the address translation can be done in parallel with the cache access. Since there is a single address space the virtual to physical mapping does not change when there is a context switch. This means that the TLB entries remain valid across context switches and a large TLB can be effectively used.

Since the Precision Architecture has a global address space, any user program can generate any address. Protection must be on a page basis. The TLB entry for a virtual page contains the physical page number as well as the access rights and access ID and tag. A TLB entry is over 70 bits long. The model 840 has one set of TLB hardware because of the TLB width, but it is pipelined and serves as both the instruction and data TLB by being read twice in each cycle. The TLB provides a physical page number to the cache which is compared with the cache tag to determine if there has been a hit or miss. The TLB hit rate is about 99.99% for the instruction TLB and about 99.8% for the data TLB.

The model 840 has separate instruction and data caches. This provides twice the cache bandwidth and allows each cache to be customized to its function. The I cache has to be very fast since there is only half of the fetch cycle to access it. This is accomplished by giving it its own address bus and putting the data RAMs on the EU board, right in the processor. The data cache has more time and can be stored into. It needs more control and is on the cache board. The caches in the model 840 are addressed with virtual addresses in parallel with the TLB lookup. The caches are direct mapped. This means that each memory location only has a single place in the cache where it can be placed. A direct mapped cache can be accessed faster than a multiway associative cache, but it has a slightly worse hit rate. The cache hit rate is about 97 to 98 percent. The data cache is store to, which means data is stored into the cache and not updated in main memory immediately. This reduces the amount of memory bandwidth required by the processor. Since DMA I/O goes to main memory, I/O buffers must be flushed from the cache before an I/O operation begins. There are special cache flush instructions used by the operating system for this purpose.

The Midbus

The model 840 has a new processor to memory bus called the Midbus. This is a synchronous bus optimized for burst data transfers of 16 or 32 bytes. It is a 32 bit bus. The model 840 has a 16 byte cache line so a cache miss can be satisfied with a single burst transfer which takes 7 clocks or 875 ns. The Midbus has a bandwidth of about 20 Mbytes per second. It is designed for reliability. Synchronous busses are inherently more reliable since there are no race conditions or synchronizer failures. The bus is fully parity protected for addresses, data, and control. In addition there are hardware time outs to detect stuck control bits. Addresses and data are transferred on the same 32 wires. The master in the transaction can arbitrate for the bus in one clock and send the address and transaction type in the next clock. The slave can cause the master to wait, or cause the transaction to be ignored and retried. The protocols and transaction types are designed to allow bus converters to be built for extending the bus if needed. The model 840 has 7 general purpose Midbus slots, 6 Midbus slots dedicated to memory, and one Midbus slot dedicated to the processor. High performance I/O cards such as a 10 Mbyte per second parallel I/O card can be plugged directly into the Midbus. Most I/O is on the CIO bus and uses the CIO channel to interface to the Midbus.

The Memory System

The model 840 has 6 Midbus slots dedicated to memory cards. The memory is built from 256 Kbit dynamic RAM chips. Single bit error correction and double bit error detection is standard. Battery backup is also standard providing power for at least 15 minutes when the AC power fails. The memory system consists of memory controllers and memory arrays. Each memory controller has 3 Mbytes of memory on it and can control a single 5 Mbyte memory array. 3 memory controllers and their arrays provide the maximum 24 Mbytes of memory. The HP9000 model 840 comes with 8 Mbytes of memory standard and the HP3000 model 930 comes with 16 Mbytes standard. The processor limits future memory expansion with new memory boards to 128 Mbytes maximum.

The CIO Channel

The CIO channel is a two board set that plugs into the Midbus. It handles DMA and polling on the CIO bus. The CIO bus built into the Model 840 has 12 general purpose I/O slots, one slot dedicated to the access port card, and one slot for the connection to the CIO channel. The CIO channel can handle up to 65000 simultaneous DMA transfers. The channel can chain multiple transfers together into a single large transfer. This helps reduce software overhead for I/O. The CIO bus and channel have a maximum bandwidth of 5 Mbytes per second. The CIO channel allows

software direct access to any CIO bus primitive operation. The HP3000 model 930 allows two additional CIO channels with two more 8 slot CIO busses in an I/O expansion box. The CIO channel has a maximum bandwidth of 5 Mbytes per second when used with the AFI parallel I/O card. CIO cards with the BIC chip such as the HPIB and serial multiplexor cards are limited to 2.4 Mbytes per second. So, the HP9000 model 840 has 2.4 Mbytes per second of I/O bandwidth for disc transfers and the HP3000 model 930 has 7.2 Mbytes per second of I/O bandwidth for disc transfers.

Conclusion

The HP Precision Architecture is a complete system solution. It includes the RISC instruction set for fast, low cost CPUs as well as hardware floating point support, a paged virtual memory system, and a new I/O architecture. The HP3000 model 930 and the HP9000 model 840 are TTL implementations for the commercial and technical markets respectively. They provide 2 or more times the performance of the previous high end machines in each of these markets.

Can distributed systems be managed effectively?

Dr. David A Thombs

Directorate General of Defence Quality Assurance
Ministry of Defence
Royal Arsenal East
Woolwich
London SE18 6TD
United Kingdom

Introduction

It is widely recognised that the combination of high speed, low cost computer hardware, reliable cheap communications and fourth / fifth generation software tools are becoming the most important factors in the design of computer systems.

However, it is also realised that the management of such systems is significantly more difficult than the single site systems of the past. The combination of a number of factors, such as, geographical distribution, communications and distributed data-base management systems and a paucity of timely information on resource utilisation and performance has made the task difficult, if not impossible.

The net effect of these deficiencies is that management decisions are being taken on inaccurate data, which can have disastrous effects in both the short and long term.

A typical small network

In order to illustrate the problems, the DGDQA network will be considered. Fig. 1 shows the distribution of current user sites and Fig. 2 shows a schematic of the basic network. The prime characteristic of the network is that it is constantly changing and tending to grow.

User capabilities on the network vary from scientists and engineers developing instrumentation and data analysis systems to clerical staff and data input staff running menu driven MIS programs.

The main user areas are:

- * Management Information Systems - Corporate IMAGE DBMS distributed over three nodes and accessed at all nodes and most terminals.
- * Technical Information Systems - Corporate IMAGE DBMS at single nodes and accessed at most nodes and most terminals.
- * Instrumentation Systems - Local use only.
- * Development - Corporate and private development, covering all areas and all nodes.
- * Other systems - Mainly statistical analysis and number crunching.

The problem

The main problem encountered when managing the type of system shown above is a lack of information on resource utilisation and efficiency. This results in considerable difficulties in making the system 'future proof'. The following areas are of particular concern but are by no means exhaustive:

*** DBMS**

Private IMAGE implementations and resource utilisation.

Corporate distributed IMAGE systems and the problems found in tightly coupled time dependent systems.

*** Communications**

Cost

Loading and congestion on private lines and public data networks.

*** Users**

Inexperienced users.

Experienced and clever users.

System response time.

*** Unattended systems**

System crashes and recovery.

Routine tasks.

*** Back up and disaster recovery**

What caused the problem.

How to recover and length of time required to recover.

*** Security**

Physical security.

Software security.

Hardware security.

Hackers.

*** Micros**

Compatibility.

Re-invention of the wheel.

Clever users

It would be difficult in a paper of this length to cover all the above areas in detail, but the paper should raise a few discussion points, and hopefully suggest a few possible lines of attack in solving the problems.

Operational problems

The term "operational problem" is used to indicate the difficulties encountered in day to day running of the system. Typical information required instantly is:

* Which users are active.

* What programs are the users running.

* Which programs is the operating system running.

Which programs are swapped out.

Which data-bases are in use.

What response time are the users getting and what is the loading on the comms lines.

For single systems the information can be obtained by the use of a dynamic system monitor, of the type described by the author at the 1984 San Jose Conference. The basic problem with this type of monitor is that, even for a single system, the amount of useful information is far greater than can be displayed on a standard 80 character by 24 line display. However, by careful matching of the system utilisation to the information displayed, a very useful operational aid can be built.

Obviously, in the case of a distributed system, the problem is far worse, due to the volume of information being multiple of the single system case. One possible solution is to use a monitor program of the type described, running at all nodes and to map the information from each node onto a separate display running on a system at the management site. This tends to produce a rather confusing set of displays and is costly on mux ports. If we consider the network above it would require 11 or 12 displays to show the activity profile. Another way of showing this type of display, would be to overwrite a single screen with the information from each node in sequence. The obvious problem with this approach is that the whole set of information is not available at any one time.

A more sensible solution to the problem is a single display showing user activity at all nodes. An example of this distributed status monitor is shown in Fig. 3. As can be seen from the example, it is easy to comprehend and assimilate and gives a very broad view of the user activity. If problems are noted, further details of the system giving problems could be obtained from a display giving more detailed information on that system.

Distributed system monitors are fairly easy to write, but a few small points should be noted.

- * It would be desirable to write a single program, running on one system, which interrogates other systems on the network to obtain the information required. This seems to be impossible because of the need to obtain information from various system tables.
- * The method used to implement the DGDQA monitor is to use a PTOP master at the network management system and slaves at local nodes. Data is passed with PTOP calls for display at the network management system.
- * The master is scheduled at suitable intervals to obtain the display updates. Care should be taken not to schedule the program too frequently, since you would end up with large comms bills and reduce the bandwidth available to users.
- * The information displayed is shown below:

US	- Two character user identifier.
LU	- Terminal logical unit number.
CAP	- User capability level.
TIM	- The number of minutes the user has been logged on.
PROG	- Program name. Due to space restrictions, CI, FMGR and QAMIS (our own user interface) are not displayed. If the user is running more than one program, only one is displayed. If for example, a user was running DEBUG on a program, either DEBUG or the program would be displayed.
ST	- Program status.
PR	- Program priority.

* For Node 1, information is displayed on the DS1000-IV comms lines utilisation. The information is shown in the following format:

LU - Comms line logical unit number
GOOD - The number of good frames received and transmitted.
BAD - The number of bad frames received and transmitted.
%ST - The percentage utilisation of the comms line since the monitor last ran.
%LT - The percentage utilisation of the comms line since the monitor started.

N.B. In the case of non HDLC cards, the information is easily obtained, since the information is contained in the system. In the case of HDLC cards the information is stored on the card and can be obtained with two XLUEX calls.

CALL XLUEX(1,CNWD,BUFF,10,1)
CALL XLUEX(1,CNWD,BUFF,11,2)

It should be noted that the bytes in the words returned are reversed and the calls only apply to DS1000-IV direct connect and modem cards. I have yet to sort out the details for the X.25 cards.

* For other points on writing this type of monitor see the 1984 San Jose Conference proceedings.

A film will now be shown demonstrating this type of monitor in action. Several points should be noted:

- * The schedule interval is small in order to reduce the running time of the film.
- * Several nodes are down in order to show limited activity.
- * The display was filmed on a live network.

Tactical Problems

The term "tactical problem" is used to indicate the difficulties encountered in maintaining efficient operation and system integrity over a period of time.

The problem can be split into two sections:

- * Control of users and resources.
- * Information on user actions and resource utilisation.

Control of users and resources

Each user of the system should have a clearly defined set of capabilities, covering all aspects of system utilisation. These should include the following:

- * Which systems can be accessed.
- * Which programs can be accessed.
- * Which segments within programs can be accessed.

- * What data can be accessed.
- * Read or write capability.
- * File creation and deletion capability
- * Peripheral access.

Management information requirements

Information required to manage the system should include the following:

- * User profiles.
- * Terminal usage profiles.
- * Program usage profiles.
- * Data-base usage profiles.
- * Unexpected event logging - program crashes, invalid log ons etc.
- * Comms lines utilisation.

Why not use the HP Accounts System ?

- * There are many problems with using the accounts system for controlling user access, the major problems being:
- * It is not capable of providing sufficient control of user capabilities.
- * Requires large amounts of memory for a large user base.
- * Lack of unusual event logging.
- * Users can not change their own passwords and the system does not enforce regular changes.
- * Little security on accounts system and a fairly inexperienced hacker can easily break the system.

How to write a better control system

There are many ways to write a better system and the following points should be considered when designing the system:

- * Use one or more HP accounts to roughly define the capabilities of your users and when logged on transfer to your control program as the primary program.
- * When users are in the control program, enter a password routine.
- * Keep routine users out of CI and FMGR.

- * Make as much of the control program as possible menu driven and do not use program names in the menu.
- * Allow users to change their passwords and enforce changes at suitable time intervals by warning users that passwords are getting old. If passwords are not changed, remove access.
- * When passwords are changed, check for surnames, first names, telephone numbers etc., this is much easier if the program can access a personnel data-base.
- * Encrypt all passwords in the data-base, using as sophisticated a routine as possible.
- * Do not link data access capabilities to people, if possible link to the position in your company. This saves a considerable effort when staff change jobs.
- * Since users are running on a small number of sessions, it is desirable to identify them to utilities, such as the dynamic system monitor. This can be done by simply overwriting the user entry in the Session Control Block. In RTE-A this can be done quite easily, in RTE-6VM it is a little more difficult but can be done without too much trouble.
- * Log all unexpected events, such as, program crashes, failures to log on etc.
- * Log all transactions to hard copy, as well as the data-base.
- * Use IMAGE to implement the data-base.
- * In a network situation how many copies of the data-base are needed? You will probably find that a copy of the data-base is required at all nodes. This introduces problems when changes occur in the data-base. If a node is down when a change occurs, the change should be made as soon as the node is back on line.
- * Look out for loopholes in programs, in particular the scheduling of programs from EDIT, BASIC etc.
- * Test and double test the system!
- * The following items could be of use in a data-base and perhaps other items as well?

Personal identifier
 Staff number
 Position in company
 Password (encrypted)
 Date password last changed
 Total log on time

Program names
 Segment names
 Data-base names
 Capabilities

LU
 Log on time
 Log off time
 Program run
 Data-base used

Systems used
Comms facilities used
Peripherals used

Time of unexpected events and description

Problem areas

One major problem is encountered when using dial up, X.25 PAD lines or a number of other devices, such as statistical multiplexors. Due to the first come, first served nature of these devices and services it is impossible to link the LU to a specific location. In the case of dial up, it would be possible to use a call back facility in which case terminal locations could be identified, but for other devices and services this is of course impossible.

Strategic Problems

The term "strategic problem" is used to indicate the difficulties encountered in the overall lifetime of a system. This covers a large number of areas, including:

- * Future loading and resource requirements.
- * Crashes and disaster recovery plans.
- * Distributed IMAGE.
- * Compatibility.

Future loading and resource requirements

Systems are constantly changing, the usage patterns alter with time and it is widely recognised that one of the most difficult management problems is predicting the future. However, using the data obtained from a comprehensive accounts system it is possible to extrapolate system usage. This is obviously prone to serious errors but is better than nothing!

Crashes and disaster recovery plans

A recent survey in the U.K. computer press indicated that less than half of all computer installations had a disaster recovery plan. This shows a remarkable lack of foresight and a misplaced trust in computers. The term disaster recovery covers a wide range of possible problems, ranging from data-base corruption problems to the loss of the entire system due to fire. It is not my intention to cover what should comprise a disaster recovery plan but merely to ask the question, " Could your company survive for days or weeks without the system ?".

Distributed IMAGE

The question of whether IMAGE is a distributed DBMS, depends on the definition of a distributed DBMS. IMAGE programs are capable of accessing data-bases on different nodes but IMAGE is not capable of spreading a data-base across nodes. The problems introduced by using distributed IMAGE, even on loosely coupled data-bases, should not be underestimated and the design should be very carefully considered with respect to actions to be performed when a node goes down. If a very tightly coupled, time dependent system is required, design with extreme care and ensure that recovery is possible, even in the event of several nodes going down.

Compatibility

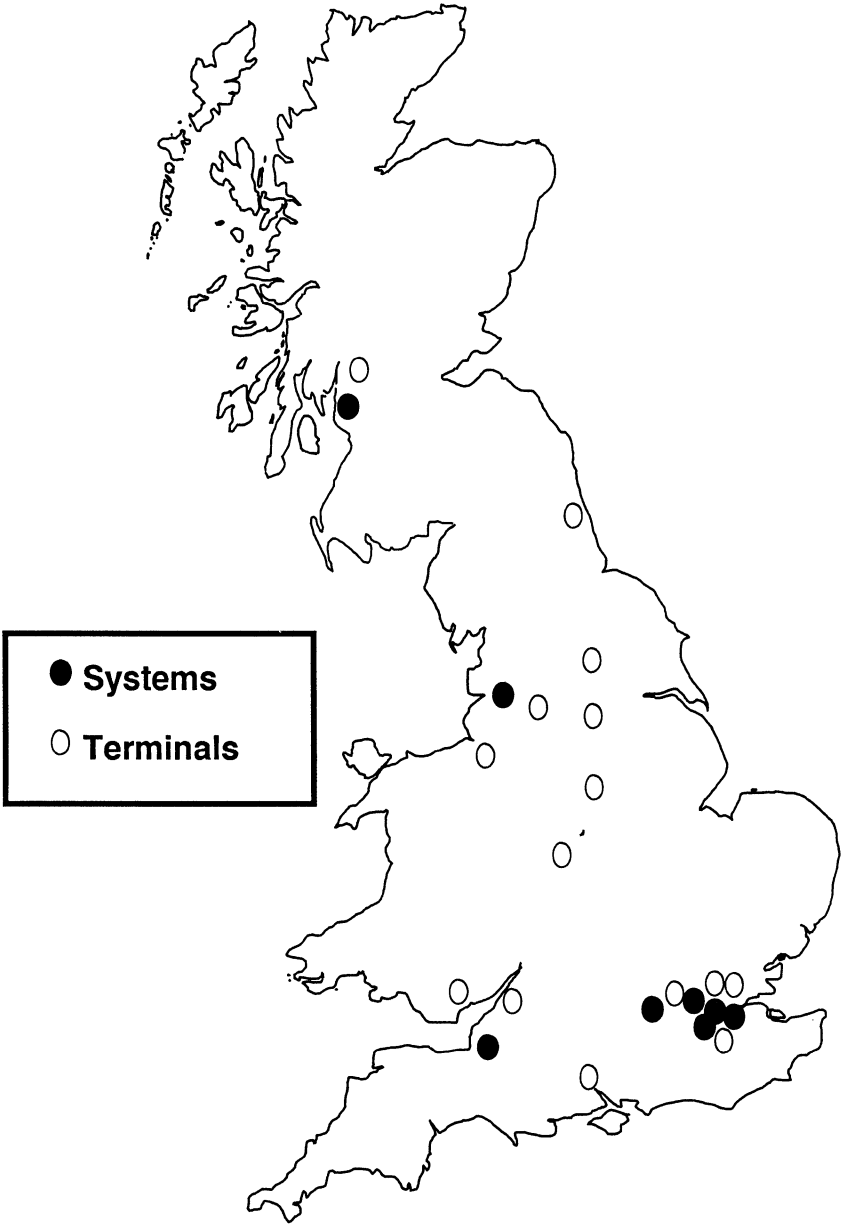
Compatibility is another serious worry and should be considered in depth. In particular the rapid expansion in the use of micros is posing a serious compatibility problem. The requirement to network the micros crops up frequently and with the mix of equipment found in the average company, this can prove to be difficult if not impossible. There are numerous other areas where compatibility must be considered but I will leave this problem to other speakers at the conference.

Conclusions

It would seem possible to manage distributed systems effectively but it is not easy. The key points are control and information. Using only HP supplied network management aids the task is difficult but by implementing suitable aids, matched to your requirements, the task becomes much easier. However, we are still in the stone age and are slowly groping our way forward. If the concept of distributed systems is to expand, it is essential that manufacturers standardise and produce more sophisticated management tools.

Copyright C Controller HMSO, London 1986

Distribution of DGDQA systems and major user sites



DGDQA Network Topology

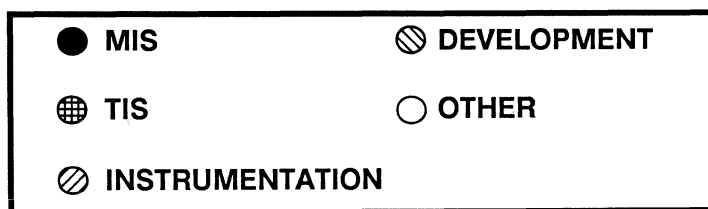
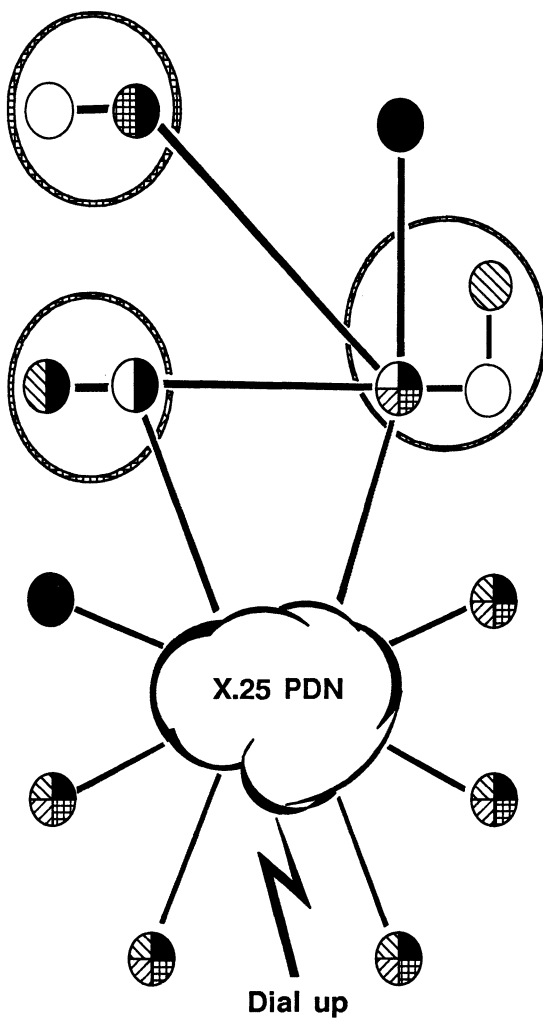


Fig. 2

DISTRIBUTED SYSTEM MONITOR

US LU CAP TIM PROG ST PR

US LU CAP TIM PROG ST PR

US LU CAP TIM PROG ST PR

NODE NUMBER 1

LU	GOOD	BAD	%ST	%LT
115	1030	1	3	1
117	703	0	41	27

AE 32	1	3	MAC32	I 99
BZ 76	1	14	MAC76	A 99
FF 91	1	10	MAC91	I 99
DP 97	63	19		
DP 96	63	35	EDI96	I 51

NODE NUMBER 2

US 1 30 68

NODE NUMBER 3 DOWN

NODE NUMBER 4

TR 66	P	62		
MA 59	N	2	SDLTR	I 90
ST 54	N	12	EDIT	I 51
DA 50	N	1		
GO 52	N	81	SK	D 99
MA 61	S	49	GRIOM	I 30
MI 52	N	13	WH	I 5

GE 63 N 2 DEBUG I 50

NODE NUMBER 5 DOWN

NODE NUMBER 6 DOWN

NODE NUMBER 7 DOWN

NODE NUMBER 8

US 1 30 68

NODE NUMBER 9

TR 71	P	150		
GF 51	N	32		
KL 63	N	69		
CD 50	N	5	PER50	I 99
ZZ 62	N	24	PER62	I 99
AA 52	N	12	PER52	I 99
DP 97	N	19		
MA 57	N	19	QUIS	I 90

NODE NUMBER 10

MA 44 63 124 EDI44 I 51

1:27 PM WED., 22 JAN., 1986

Fig. 3

ESTABLISHING A SUCCESSFUL HP1000 CONSULTING PRACTICE

A. Marvin McInnis
5250 W. 94th Terrace, #114
Prairie Village, KS 66207

INTRODUCTION

Independent consulting can be an attractive alternative to more traditional career paths for experienced HP1000 specialists. While many mature HP1000 users are technically qualified to be independent consultants, technical competence alone will not assure success. Success in professional consulting requires business, marketing, and communications skills as well as technical knowledge. And just as important, some very talented HP1000 specialists are unlikely ever to be successful in private practice due to personality, lifestyle, or other factors.

This presentation assumes that you are interested in establishing yourself as an independent consultant specializing in HP1000 systems. We will examine some of the essential elements of building a successful practice, and while the major emphasis will be on HP1000 consulting, the general principles will apply to consulting in other technical areas as well.

THE PRACTICE OF CONSULTING

We have all heard humorous definitions of the consultant; my favorite is "anyone who is out of work but has a briefcase." For our purposes, however, we will define a consultant as a person with specialized skills or knowledge who provides advice and/or services, for a fee, on a contract basis.

Although there are many types of consulting organizations, from the job shops to the large professional (accounting, engineering, etc.) and consulting firms, we will assume that you will be establishing a private consulting practice or small consulting firm. These businesses (and they are, first of all, businesses) can range in size from a single individual to more than twenty consultants, are privately owned, and are often entrepreneurial in their orientation. Consulting in specialized technical areas is their primary business activity, and most clients are small to medium sized businesses or government agencies. Projects can last from less than one month to more than a year, and the firm will typically perform work for several clients each month.

Establishing a Successful HP1000 Consulting Practice

ADVANTAGES/DISADVANTAGES OF PRIVATE PRACTICE

As with many other types of entrepreneurial businesses, consulting in private practice offers independence and flexibility among its advantages, and there is virtually unlimited opportunity for professional growth. There is also the POTENTIAL for financial reward significantly greater than can be expected as someone else's employee. It has been my observation that most entrepreneurs who stick with it for more than ten years are far better off than if they had remained working for someone else. But first you must survive the crucial start-up years!

Private practice can be a particularly attractive option for experienced technical specialists who are, for some reason, blocked in their career development. It also can be a good choice for women, minorities, or otherwise competent persons with limited academic credentials.

As an independent consultant, your reputation is directly related to the perceived quality of the work you perform for your clients. No one else will receive the recognition for your accomplishments.

As an added bonus, relatively little start-up capital is required to establish a small consulting practice.

Just like any other entrepreneurial venture, the biggest disadvantage of private practice is the risk of failure. There is no guarantee of success and the initial risk is always high.

And, as an independent consultant, you must forever give up the myth that anyone but you is responsible for your success or failure. This attitude is very difficult for some persons to adopt, but I believe that it is essential to the long term success of any start-up business.

Another disadvantage is that you will give up the apparent security of regular employment; if you do not work you do not eat. However, I feel that this "security" is highly overrated since in the past decade even very large companies have laid off substantial numbers of professional and technical employees.

In the early years as an independent, your income will almost certainly be irregular, and this can be a source of considerable stress. Some persons can never adjust. And even when your practice has achieved financial stability, you will still find it much more difficult to borrow money (for any purpose) than when you "had a real job."

In private practice, you will accrue some long term financial and contractual obligations from which you cannot just resign, as you might from a regular job. Working for yourself is usually more stressful than working for someone else, and the effort necessary to achieve success in your work may be highly detrimental to your personal and family life.

Finally, there is always the risk of developing a successful practice and then discovering too late that the state of the art has passed you by. The risk of dead-ending yourself is always there, just as in conventional employment, and the results can be just as tragic. Over-specialization will substantiantially increase this risk, but we will discuss that in more detail later.

WHO SHOULD/SHOULD NOT CONSIDER PRIVATE PRACTICE

Qualifying yourself is one of the most important steps prior to establishing a business. Personal preferences, lifestyle, past work experience, and professional abilities are all important factors in this self evaluation. While by no means complete, the lists below contain some of the things I feel are very important in considering private practice.

You may be a good candidate for your own practice if you posses most of the following qualifications:

- 1) You consider yourself very competent, with at least five years of work experience in your technical specialty and five years with HP1000 systems.
- 2) You are a generalist rather than a specialist. You get extra points if your academic training was in something other than computer science.
- 3) You have an entrepreneurial personality, and have had since you were a child. (Was it usually YOUR idea to open a lemonade stand or did you always help someone else?)
- 4) You are self-motivated and well organized, and often impatient with your employer's formal procedures or with the office bureaucracy. You are sometimes told that you try too hard or that you care too much about the quality of your work.
- 5) Beyond being technically competent, you can communicate clearly in standard English, both

orally and in writing. Remember that listening effectively usually is as important to communication as being able to express your own ideas.

6) Persons in your present organization usually seek YOU out for assistance with their technical problems, rather than going elsewhere.

7) You are analytical, holistic, and creative, and can be these things all at the same time.

Conversely, you should consider yourself a poor candidate for private practice if you recognize many of the following characteristics in your personality:

1) You are a perfectionist. Perfectionists often have great difficulty finishing anything, which is seldom what the client wants.

2) You place a high value on security and stability in all aspects of your life, financial or otherwise. You do not function well in ambiguous situations.

3) You must depend on a regular income to meet your financial obligations. (You will probably have NO net income for the first year to eighteen months of private practice!)

4) You place a high value on your marriage and family, or on other aspects of your personal life. (The available statistics on this subject are rather grim.)

5) You work best doing one task at a time, and find it difficult to handle more than two or three things at once.

6) You are basically an introvert and almost always do your best work when assigned a task and then left undisturbed.

7) You think that selling is a disreputable activity, or at best a waste of valuable time. (You will have to develop marketing and sales skills to succeed!)

TEN REASONS WHY NEW BUSINESSES DO NOT SUCCEED

Before going any further, it is worth considering why more

than four out of five new businesses fail within the first two years. Here is a highly abbreviated list of ten major reasons for this dismal record:

1 - Lack of Commitment

This is a personal matter. If you are going to establish a practice, make a commitment! Don't do it half heartedly. I recommend that you not start on a part time basis, because doing so fosters the attitude that "I'll try it for a few months, and I can always find a job if it doesn't work out." You must be determined in order to succeed!

2 - Inadequate Capitalization

This is clearly a financial problem. Realistically, it will probably be eighteen months before your consulting practice will generate enough income to pay the business overhead expenses, taxes, and a regular salary to you. And even then, capital will be required to fund growth.

3 - Too Few Clients

This is primarily a marketing problem, but may also reflect on management. If you do not develop a large enough, and somewhat diversified, clientele you will certainly fail.

4 - Lack of Technical Competence

Large companies may occasionally get away with poor technical performance, but your practice cannot. Indeed, competence is often the primary reason that a client retains a small firm, rather than going elsewhere.

5 - Overspecialization / Underspecialization

This is both a technical and management problem. Increasing specialization decreases the potential number of clients for your services, but may also increase your value to those remaining clients. Determining the appropriate degree of specialization requires both a careful evaluation of your capabilities and an understanding of your potential market.

6 - Lack of Persistence / Inconsistent Goals

This is clearly a management problem! Very few businesses can survive if allowed to wander from project to project without any underlying direction. You must decide what you are NOT going to do as carefully as what you ARE, and then stick with your plan for a reasonable time.

7 - Poor Management

The long term survival of your practice depends on good day-to-day management. The firm must have long term goals, and some kind of plan for achieving them. Daily decisions must then be evaluated against your goals: "Does this decision move us in the direction of our long term goals, or in some other direction?"

8 - Overcommitment / Too Many Clients / Projects Too Large

This is both a financial and a management problem. After the critical start-up phase, the most dangerous period for any business is when it begins to be successful and is experiencing rapid growth! Financial capability and management decisions must control the growth of your practice, not the other way around.

9 - Poor Client Relations

If your clients are not satisfied with your services, for whatever reason, they probably will not be back. This is ultimately a management problem, even though it may have its roots in technical, ethical, interpersonal, or marketing areas.

10 - Stagnation / Lack of Technical Growth

This is one of the least acknowledged management problems. It is often quite difficult to reserve a substantial portion of your time for learning, research, and professional development. But always remember that you must continually be preparing for next years' projects.

DESIGNING YOUR PRACTICE - DEVELOPING A BUSINESS PLAN

While it may seem overly basic, before doing anything else you will have to decide:

- what services you will offer
- who the potential clients for your services are
- what geographic area you will serve
- how you can reach your potential market
- how you can do all this profitably

These basic decisions will become the core of your business plan. While I am not an advocate of the formal business

plans so favored by MBAs and bankers (who, I suspect, seldom read one in its entirety), I do believe in working up a comprehensive outline of the new business for one reason: the evidence convincingly supports the notion that the clearer an image you have of your goals, the more likely you are to achieve them!

Let's examine some of the elements of a business plan in more detail.

THE SERVICES YOU WILL OFFER

I take it as a given that you are competent and technically qualified to be an independent consultant. But never forget that the very best work you can do is barely adequate when working in private practice.

For a small business, especially a consulting practice, to be successful, you must carefully select a market niche and commit your resources to servicing it. You may ultimately choose several market niches, but each of them must be chosen carefully. There are good business opportunities everywhere, and often the most difficult decisions are in choosing what you are NOT going to do.

In defining the scope of the services which you intend to offer, you will ultimately have to deal with the twin issues of expertise and specialization. Obviously, as an established wizard, the HP1000 will be one of your specialties. But will you try to cover the full HP1000 product offering, or will you limit yourself to the subsystems with which you have the most experience? Or will you specialize in a particular application area? Will you limit yourself to HP1000s or will you work on other HP products? And what about other vendors' systems?

The advent of the Spectrum-family technical computers offers an example of the choices available. I predict that these new systems will offer numerous consulting opportunities in at least three distinct market niches:

- 1) first-time HP technical computer customers
- 2) existing HP1000 customers who are migrating their applications to the Spectrum family
- 3) existing HP1000 customers who are stampeding off to other vendors rather than migrate

In defining the services you will offer, don't forget the intangibles. For example, HP has been criticized for the

lack of experience of many of their SEs, as well as for the unavailability of the more experienced SEs and Specialists; indeed, HP often goes to great lengths to isolate the SEs and Specialists from their customers. You can compete effectively with HP by offering what they do not: both expertise and availability. (Note that HP does consider itself one of your competitors; this was not always so, I'm sad to say.)

But defining the services you will offer is only the beginning - you must also have clients for those services.

MARKETING YOUR SERVICES

Defining the services you will offer, identifying potential clients, and making them aware of your services are the central functions of marketing, which is a business activity distinct from sales. Marketing is such a large and important topic that it is really beyond the scope of this presentation, so we will consider only a few central issues most pertinent to private practice.

The importance of marketing to your success cannot be overemphasized; poor marketing is second only to inadequate capitalization as a cause of small business failures. This can be a particular problem for the independent consultant, not only because marketing is probably the most "foreign" of the skills you will have to learn, but also because you have nothing to sell other than your own expertise.

From the beginning, do not deceive yourself by thinking that your potential clients will seek you out. This is a mistake that too many technical people make; the world just doesn't work that way! It is your responsibility to locate your potential clients and make them aware of your services. This is hard, often discouraging work; you will be doing very well if more than five percent (one in twenty!) of the prospects you contact ever retain you.

The first task of marketing is identifying your potential clients. Obviously, your market will come from somewhere within the body of HP1000 users, those whose applications roughly match the services you are offering. But you will need to focus on some smaller number of prospects. Additional selection criteria might include:

- What geographical area will you serve? You will probably have to serve more than just your community unless you are located in a large metropolitan area with many HP1000 users.

- In what industries or application areas (manufacturing, laboratory, communications, etc.), if any, do you wish to specialize?

- Should you concentrate on small-to-medium sized businesses, large corporations, government agencies, or some mix? Your personal background may provide you with a particular marketing advantage with certain groups.

You should continue in this way until you have developed the selection criteria for your primary market. You may also have identified one or more secondary markets as well.

A companion task is to identify the specific companies which satisfy your market criteria. How do you do this? Research! That is, research and plain hard work.

Although HP policy treats customer lists as confidential, you can certainly identify some HP users in casual conversations with HP employees. Interex and local users groups are also a good way of identifying prospects. Local newspapers and business journals are sources of information, as well as the business directories available at most metropolitan public libraries. "Help Wanted" ads are a good resource; an advertiser may not have considered consulting as an alternative to hiring.

You should also try to identify at least one contact person in each prospect organization. This person should be someone who has technical responsibility and the authority to retain your services, but any name is better than none. If you do not know where else to find out, business directories often contain this kind of information, although it is often obsolete.

Identifying your prospects alone does no good at all if you cannot reach them and make them aware of your services. While the solution in most businesses is "advertising," the market for your consulting services is so small that traditional advertising approaches are usually inappropriate. (COMPUTER CONSULTANT, a trade journal, reported that fifty percent of consulting firms responding to a survey on business practices had never advertised!)

Probably the most effective technique for your initial contact is a standard brochure mailing to your prospects. Your standard brochure should be a simple document which introduces your firm and describes the services you offer, with emphasis on special areas of expertise. A simple 3 x 9 inch format (an 8 1/2 x 11 inch sheet, folded twice) is

convenient and easy to mail, although you may prefer a larger and more elaborate presentation. Regardless of the format you select, the brochure must be carefully written and must look professional. Since this brochure will make the first impression with many of your prospects, it must reflect the very best work you can do! If your research has yielded the name of an appropriate contact person for a prospect, include a short cover letter introducing yourself; but form letters are forbidden!

It is an established principle of advertising that the more frequently prospects are exposed to a product name, the more likely they are to try it. Using several quality marketing techniques at the same time can increase your overall marketing effectiveness. Some techniques which have been used successfully include:

- conducting seminars
- writing articles for publication
- presenting papers at professional conferences
- public speaking (through speakers bureaus)
- publishing a newsletter
- lecturing or teaching courses at local colleges

In addition to these techniques, by far the most effective marketing tool available to you is word-of-mouth referrals from your existing clients or from third parties such as HP employees. These referrals not only identify the potential clients for you, but they also contain an implied endorsement. By all means solicit referrals from your existing clients and HP, but do not depend on them for your success.

Whatever marketing techniques you choose to employ, remember that quality is of paramount importance. You must always be a professional, and you must always project the image of professionalism that is characteristic of your practice.

FINANCIAL CONSIDERATIONS

The final aspect of your business plan is financial: can you operate your consulting practice profitably? This is the point at which you must suppress your desire and determination to succeed and evaluate the situation as objectively as possible.

One of the advantages of consulting as a business venture is that relatively little start up capital is required. Balanced against that, however, is the fact that you have nothing to sell but your own professional skills. At the minimum, you will need sufficient initial capital to pay the business overhead, your own personal obligations, and taxes for at least eighteen months, until your practice becomes profitable.

Before you can determine how much start-up capital your practice will require, you must develop the most realistic possible estimate of your anticipated income and expenses. But always remember that these figures are just that: estimates! Your estimates will almost certainly be wrong, but you must do the best you can and then adjust them for contingencies.

You should select your business form and accounting method at this time, since these will help in identifying the dozens of categories of income and expense which you must consider.

As a business form, Sole Proprietorship is certainly the easiest and least expensive to establish if you will be the sole owner of the company; you can always convert to a small business (IRS Subchapter-S) or regular corporation at any time in the future. You definitely should discuss this decision with a business consultant; this may be your attorney and/or your accountant, but remember that they may have a conflict of interest since incorporation usually means more business for them!

In selecting an accounting system, I would recommend that you use the accrual method rather than the cash method, although some sources will disagree. It is my feeling that the accrual method, while slightly more complex, is more flexible and can give you a more accurate picture of the monthly performance of your business. But regardless of the method you choose, your accounting system MUST be formally set up and faithfully maintained.

I would also recommend that you plan to do the bookkeeping yourself for the first couple of years, with periodic review by your accountant. You need to develop an intuitive understanding of the accounting transactions in your business; it is in your interest to do so, and besides, it is your legal responsibility that your books accurately reflect the conduct of your business!

Two fundamental questions at this point are how to charge for your services and how to set your rates. At the most

basic level, you can bill by units of time (hour, day, etc.), a flat fee for a specified job, or on a retainer basis. Unless there are special circumstances, I recommend that you charge by units of time whenever possible. It took me many years (too many!) to figure out that my most satisfied clients, and the ones I was most satisfied working for, were my hourly clients. But each client is different, and you will have to decide for yourself.

Regardless of how you bill, your charges must ultimately be based on some hourly rate for your time. What should your hourly rate be? You will have to decide for yourself, but here are some things to consider:

- It is a disservice to yourself and to your clients to charge so little that your practice cannot be profitable. If it is not, you won't be in business very long.

- It has been my experience that, given the time that the management of your practice will require, a maximum of 60% of your working time will be billable to your clients. Your average billable time will be less.

- Your business overhead expenses, plus a reasonable salary for yourself, will determine the break-even point for your practice. The absolute need to eventually be profitable will point toward the lowest rates you can charge.

- HP charges \$825 per 8-hour day for SEs and \$1,000 per day for Specialists. This probably sets the upper limit for the rates an independent can charge, regardless of your skills. (We won't even consider the "big name" accounting firms which are charging \$150 per hour or more for "consultants" with less than six months experience!)

Your overhead expenses will be somewhat easier to estimate than your income, but still I suggest that you develop your best estimate and then add 25% for the inevitable contingencies. At a minimum, your overhead estimate should include the following categories of expenses:

- Salaries and employee expense (include yourself)
- Professional services (accounting, legal, etc.)
- Office and facilities (office rent, utilities,

telephone, office supplies, etc.)

- Marketing
- Professional development (continuing education, professional organizations, reference materials, etc.)
- Business travel and subsistence
- General overhead (insurance, taxes, interest)
- Equipment rental and maintenance
- Depreciation (capital items)
- Miscellaneous

Once you have a reasonable estimate of your income and overhead expenses, you can determine how much start-up capital will be required. This is something that you should review carefully with your accountant at this time.

The start-up capital you need is not usually available from conventional financing sources due to the intangible nature of consulting services; you will have to locate the sources for yourself. But regardless of the source, this capital should be in cash or some other form which can readily be converted to cash as needed. Some possible sources include:

- Personal savings
- Life insurance cash value
- Stocks and other investments which can easily be liquidated
- Second mortgage or refinancing on your home (You must do this while you still have a job, and you must not mention that you plan to go into business for yourself!)
- Loans and/or equity investment from private sources (I DO NOT recommend family or personal friends!)
- Loans and/or equity investment from potential clients

Note that because your principal asset (your expertise!) is intangible, you probably will not be able to borrow money to

finance your business. (Your bank won't even want to talk to you for the first two years of your practice. And forget about buying a home if you don't already own one!)

If the capital you are able to obtain is inadequate to fund your business plan for at least eighteen months, you must go back and develop a revised plan. You may have to do this several times before you end up with a plan which has a reasonable chance of success.

When all is done to your satisfaction, have your accountants review the financial aspects of your plan with you. They can give you their advice, but remember that the final decisions are YOURS.

HINTS FOR MANAGING YOUR PRACTICE

Having gone through the admittedly tedious process of developing a business plan, you are much better prepared to succeed than 90% of new businesses! Now, in addition to the factors we have previously discussed, your success will depend on your ability to effectively manage your practice on a day to day basis.

Here are some hints, in no particular order, which may be helpful to you:

Three of the most important persons in the success of your practice are your attorney, your accountant, and your banker. Choose them as carefully as you would choose a lover, and take the time to determine that they are as competent in their fields as you are in yours. You will not need them often, but mutual confidence and understanding will be very important when you do.

[If you can, find a banker who will establish a credit line of \$10,000 to \$20,000 for your business, borrow against it, and put the money into a CD. You will end up paying about 2% to build yourself a credit history this way; it's one of the best investments you can make! (Note that even if you incorporate, for the first few years your personal guarantee will be required on loans to the company. Other creditors may require your guarantee as well.) Incidentally, NEVER pay a credit-line loan off in full before it matures - pay it down to \$1 instead. This will drive your bank nuts, but it will keep your credit line open!

Maintain a daily awareness of your business' cash position. Daily cash management is often crucial when cash is a scarce commodity, as it surely will be at times during your first year. Make it a policy always to pay your bills on time;

your dependability can be a valuable asset when dealing with creditors.

Maintain adequate insurance to cover your obligations, but do not over-insure. Note that professional liability insurance is virtually unobtainable, even though your exposure is very low. Consult with your attorney (BEFORE you have a problem, please) if you are concerned about professional liability.

Avoid becoming dependent on a few big clients! Diversify both your clientele and, if possible, your clients' industries. For example, many consultants working exclusively in the petroleum industry are in serious trouble right now.

As a matter of policy, never undertake work without some kind of written agreement, especially with new clients. This agreement can take the form of a purchase order issued by your client, a standard form you provide, a formal contract, or a simple letter of agreement. Whatever the form, the agreement should include the following:

- a description of the work to be performed, in sufficient detail to provide a basis for determining your satisfactory performance
- a statement of the fee you will receive for your work and how it will be paid
- a statement establishing that you are an independent contractor rather than an employee of your client (consult your attorney for standard language acceptable both to the IRS and your state government)
- a statement of mutual confidentiality with respect to trade secrets and business practices
- if appropriate, a project schedule
- if the project involves creating new software, a clear statement of ownership and license terms
- if appropriate, a statement of how changes or additional work beyond the scope of this agreement will be handled

Normally, you can safely perform hourly work for an established client without a written agreement. But beware of the new client who expects you to work on the basis of an

oral agreement and a handshake; there are almost certain to be problems. A written agreement provides protection for both parties, as well as a clear description of the job to be done.

Always identify a person in the client organization who has both the willingness to work with you and the authority (the power!) to make things happen. Note that this person may or may not be your designated contact with the client, particularly if it is a large organization.

Never accept a contract in which you expect to lose money or just break even, especially with the expectation that doing so will generate future business. You must be profitable to survive, and it is a disservice to your other clients to hasten your own demise in this way. By all means, donate some of your time to charitable work as a matter of professional ethics, but do not confuse charitable work with business.

Have a basic contingency plan for the business, in case things go badly. Decide when you will throw in the towel, should it ever become necessary. Establish regular check points (at least every six months) to re-evaluate the progress of the business, make changes, and even modify the business plan if necessary. Remember that, when the chips are down, doing without is an excellent survival tactic.

YOUR RELATIONSHIP WITH H-P

While HP doesn't have a formal consultant program for technical computers, as with their commercial systems, they still can be a valuable source of leads and referrals.

As a matter of policy, HP is very conservative about referring their customers to consultants. This has always been so, but now that the Systems Engineering organization is competing for some business formerly referred to third parties, they are being even more selective. But HP still considers independent consultants to be an important resource in support of their sales activities.

HP sales and support personnel are outstanding in the industry, both from a technical and an ethical standpoint, and working with referrals from HP does not present any problems as long as you both remember that your primary responsibility is to the client, and not to each other. Though it occasionally causes conflicts with HP, your recommendations must remain independent and professional. Even when you must disagree with HP, you can offer your opinion without being critical of them.

Finally, don't expect HP to find work for you. Rather, position yourself to be an additional resource HP can call on to solve the customer's problems. Calling your local sales rep every week to solicit referrals is the surest way I can think of to kill the golden goose.

MAINTAINING GOOD CLIENT RELATIONS

Maintaining good client relations is essential for your long term success. Fortunately, it is not difficult to do, but it does take conscious effort on your part. The reason that you must work at client relations is that it is based on how the client perceives your work, which may be different than how you perceive it.

Obviously, doing your best work is a key component of good client relations. Correspondence, reports, documentation, and programs should always project the best work you can do. Documents must LOOK good before the first word is read! A corollary to this is: don't let a client see your poor or incomplete work. Ever!

Always strive to project the image of the competent, dignified professional that you are. Dress slightly more formally than the norm for each client, but always maintain a natural look. Less formal dress may connote sloppy work, while much more formal attire may project a stuffy or superficial image.

Effective communication is also very important. It is easy to make your work mysterious and arcane, but your clients will usually be delighted if you take some time to include them in what you are doing. I find that I spend a considerable amount of my billable time teaching, with the goal that the client will become less dependent on me, and this has been well received.

Don't be afraid to tell a client "I don't know the answer to your question, but I can find it for you." Some consultants strongly disagree with this advice, but I feel that it is not your job to know all the answers; rather a major part of consulting is knowing where to find the answers, and developing solutions where no definitive answers exist.

Here are several more suggestions for maintaining good client relations:

- As a general rule, I recommend that you do not mix personal and professional friendships.
- Maintain absolute client confidentiality, even

when it is not required. And even when you have your client's approval, be discreet.

- Never speak disparagingly of another client, a vendor, or a competitor.

- Establish a standard fee schedule. Avoid charging clients different rates for the same kind of work. While this may or may not be unethical (I feel that it is), it is certainly asking for trouble!

- If you maintain a good professional relationship with your clients, collections will probably not be a problem. But don't be bashful about making a polite inquiry if a payment is late.

PROFESSIONAL ETHICS

Contrary to cynical observations by some people, it is possible for a business to operate ethically and still be competitive and profitable. It is fortunate that ethical problems are a fairly infrequent occurrence in consulting, since when it comes to maintaining ethical standards you are pretty much on your own.

The ethical problems a consultant does encounter are seldom black and white. In fourteen years of consulting, I can think of only two cases of blatant ethical conflicts. (One involved a client who wanted a highly non-standard transaction embedded in an inventory system, and the other involved a vendor sales rep - not HP! - who was insistent that my firm receive a "commission" on a potential sale.) A more common, and much more subtle, area of ethical problems is conflict of interest.

Unfortunately, it is my experience that you cannot look to established professional organizations for much in the way of ethical guidance. Many of them do publish canons of ethics which are worthy of your consideration, but enforcement of those standards has been, at best, inconsistent and ineffective, and at its worst, petty and ridiculous. (One example sticks in my mind. In the fall of 1983, the National Society of Professional Engineers reprimanded an engineering firm for having its name on the uniforms of an amateur softball team, having concluded that this was unethical conduct. That same month, the entire country was shocked to learn that more than twenty Professional Engineers had been indicted for bribery in the state of Maryland, along with the Vice President of the United States!)

My conclusion is that you are going to have to establish your own ethical standards, perhaps incorporating some of the better ideas from outside sources. Always strive to maintain absolute honesty and independence of your opinions, and to deserve the confidence of your clients. Some specific suggestions are:

- Be constantly alert for potential conflict of interest. When confidentiality permits, discussing the potential conflict with all parties is usually the best way to resolve it.
- Never accept commissions or finders fees from vendors. This is risky even if you do so with the full knowledge of your client, since it may impair your independence.
- As a general rule, do not sell proprietary products, or products in which you have a financial interest, to your consulting clients. It is difficult to remain objective as a consultant if you are also trying to sell hardware or software to the same client.
- Always give credit to others for their work rather than claiming it as your own. You are just as valuable to the client for knowing where to find it.

CONCLUSION

Like anything in life worth achieving, establishing your own consulting practice takes hard work and dedication. Over the long run it can be very rewarding, both in financial terms and in personal satisfaction.

I hope that this presentation has provided information you will find useful in determining whether or not private practice is a career alternative for you. And for those of you who choose this road less travelled, I wish you success.

Databases in the Scientific and Engineering Community

Husni Sayed
IEM, Inc.
P.O. Box 8915
Fort Collins, CO 80525

INTRODUCTION

A database is, in the broadest of terms, a collection of information, usually related in some way. Databases have long been very popular in the business world due to their ability to store, manage and process very large amounts of information. Recently, other disciplines, including those that are Science and Engineering related, have begun to realize the potential of databases. As easily as a database manages mailing lists and inventory, it can manage numerical data gathered from a variety of sources.

Though databases are used in very diverse fields, there are many differences in what a business person would need from a data base, and what a Scientist would need. Therefore, before choosing a database, you must be aware of what is available, and what features you need.

HIERARCHICAL VS. RELATIONAL DATABASES

Among the ways in which databases can be structured, hierarchical and relational data bases are the most common types. These two different ways of structuring a database support very different applications.

A hierarchical database is organized into different "layers" of information, like a tree structure. When you are searching for a particular piece of information, the length of time it takes to locate that information depends upon where in the hierarchy it exists. To access information at the lowest level, you will need to scan through all the higher levels first. Also in a hierarchical database, every database is completely separate from the others. The separate databases have no way to share information, as only one data base at a time can be accessed. This can mean a great deal of redundancy if different databases have a lot of information in common. If you want to redefine relationships in a hierarchical database, the entire structure of the database must be changed.

A relational database permits more flexible handling of data, because data items can be related in many ways. A relational data base is structured more like a multi-dimensional table, where a single piece of information can be identified by many different attributes. Data organization in a relational structure can be re-defined without changing the data structure, rather by simply changing or adding relationships between data. A specific

piece of data is easily accessed, without searching through every piece of information that is located physically before it. Also in a relational system, information can be shared between different databases, eliminating redundancy. The major drawback to a relational system is that it appears to occupy more space than a hierarchical system as the relations get more complex and the size of the database gets smaller. In a very large database, a relational system may be smaller than a hierarchical system, due to the elimination of redundant information.

MENU DRIVEN VS. LANGUAGE DRIVEN

Another way to differentiate between databases is to classify them as menu driven, or language driven.

A menu driven data base tends to be very user-friendly, and easy to use. As with any menu driven program, a menu driven database guides you through the data base. The database modules list your options, and wait for you to select the option you need. While menu driven systems are initially very easy to use, more experienced users may find that they are too limited in their capabilities. Some menu driven databases offer script capabilities, which is their way of allowing the experienced user to automate their data base access. Script capabilities allow you to enter sequences of keystrokes to be stored on disk, and later retrieved and executed.

A language driven database, on the other hand, is very flexible, but may initially be very difficult to use. Such a system requires that you learn a new language to use the database. Rather than having your options displayed on the screen, they are listed in a user's manual--and you must program them in to use the database. Especially for those who dislike programming, this may be a very negative characteristic. However, language driven databases may prove to be more flexible than menu driven ones. If a menu driven database provides other "language" capabilities, such as a programming interface, the menu driven database may be more flexible and efficient than the language driven database: especially if the programming interface allows the use of common high-level languages, such as FORTRAN or Pascal.

DATABASE NEEDS OF SCIENTISTS AND ENGINEERS

Once you know a little about databases, you need to decide on the characteristics that are important to your needs. A few characteristics that are important to every application are:

- Speed
- Flexibility
- Ease of use

Obviously, no one want to sit around waiting for their database to process information. Flexibility is important for two reasons: first, you must be sure that the database can be structured to fit your data; second, it must be able to manipulate or analyze the data in the way that you want. Ease of use is always important, lest you get frustrated with the system, and go back to your calculator and pencil.

The most common ways in which a database can be used by Scientists and Engineers can be grouped into 4 main categories: data acquisition, data manipulation, data anal-

ysis, and data presentation. Data acquisition entails the gathering of information, from any source (perhaps an instrument). Databases can be very efficient tools for the data acquisition process, as long as the user is able to collect the data programmatically (automatically) rather than manually. Data manipulation entails storing, grouping, and re-grouping data. This sort of manipulation is done more easily and quickly by a relational database than a hierarchical database. Data analysis can be done easily by any database that incorporates statistical analysis capabilities, and data presentation can likewise be very efficient if the database you are using supports integration of text and data into forms and reports.

ADIMENS AS A SOLUTION

Features of ADIMENS

ADIMENS is a successful, time-proven relational database that is currently licensed and marketed by Hewlett-Packard in Europe. ADIMENS has many features that make it an attractive product for Scientists and Engineers:

- ADIMENS is a relational database, allowing for greater flexibility, and eliminating repetitive information.
- With a guaranteed record access time of less than 0.1 seconds for HP Series 200/300 users, ADIMENS is extremely fast.
- ADIMENS allows for menu driven, script, and programmatic access into database files.
- ADIMENS removes system dependence. Available on a wide variety of HP computers, (HP 110+, 150, 1000, 3000, 9000 and VECTRA), applications can be developed on a Series 300 under Pascal, then run on a VECTRA under MS/DOS, on a Series 300 or 500 under HP-UX, or both.
- ADIMENS includes a fully integrated package for statistical analysis of data.
- ADIMENS can be used with other application packages and programs, such as ME10 and EGS. It is very suitable for use by OEMs.

Components of ADIMENS

The ADIMENS relational database is composed of four segments, or modules: INIT, EXEC, TEDI and PROG.

INIT is the module that is used to design your database. Based upon the type of data that you will be gathering (or have already gathered), you should decide on a layout for your database, and how different data items will be related. After these decisions have been made, INIT allows you to create files and fields for your database, create screen masks for your database (to help you access and/or view your data), set access rights on items in the database, and change the storage size for key files and data files. INIT can be menu driven or script driven.

The EXEC module can be used after the database has already been designed. It allows you to enter, change, find, display and remove data. With EXEC you can also select data items based on up to 150 different criteria (characteristics), and set up free-form spreadsheet calculations, obtain information on the status of the database, or perform statistical analysis of a data set. The statistical analysis segment of this module is flexible enough to allow you to define (and save) your own analytical equations/operations. This segment also allows you to select groups of data on which you would like the analysis performed. EXEC can be menu driven or script driven.

TEDI provides word processing capabilities that allow you to access, and incorporate, information from the database. This can be extremely helpful for generating letters, forms and reports that include your data.

PROG is the module that allows programmatic access into your data base files. PROG is a library of pre-defined procedures that can be accessed from FORTRAN, Pascal and C programs. This incorporates the flexibility of language-driven databases, without making you learn a new programming language. The PROG database interface establishes a link between your program and the database. The pre-defined procedures in PROG allow you to write application programs that can enter information into the data base (perhaps in the form of measurements read from instruments), access and manipulate data already contained in the data base, analyze data, produce graphical output, etc. In fact, any function of the database can be accessed programmatically using PROG.

Most of the uses for databases in the Scientific and Engineering communities are in 4 areas: data acquisition and management, data manipulation, data analysis, and data presentation. Now we will see how ADIMENS can help solve those problems most often faced in these areas.

Data Acquisition and Management

Data acquisition occupies a large percentage of any Scientist or Engineer's time. With many database systems, you could spend time acquiring data, and then spend an equal or greater amount of time entering the information into the database. This highlights one attraction of a database that allows programmatic access into its contents. With such a system, you can write a single program that will collect your data, and enter it directly into the database. Of course, before you can enter information into the database, you must know how your data is going to be organized. This is done (either directly, or programmatically) using INIT.

Once you have acquired your data, it needs to be maintained, or managed. Data management involves maintaining necessary relationships between data items, and changing or updating the data as necessary. Another outstanding feature of ADIMENS is that it allows you to change or add fields to data records, without destroying the information you have collected. In many systems, you cannot restructure your data without destroying it.

Data Manipulation

Data manipulation is the process of reordering or changing your data to reflect a particular situation or condition. As mentioned earlier, reorganizing data in a hierarchical database can be very time consuming, as each reorganization entails restructuring the

database. (Not to be misleading, the restructuring is done by the computer, not by you.) And if you want to group together data based upon information contained in more than one database, forget it. At least let your spouse know that you won't be home for dinner

The ADIMENS database makes it very easy to sort your data in different ways. In fact, in many instances, it is faster to produce sorted than non-sorted output. This process is faster than hierarchical systems, because no restructuring of the database is required: all of the necessary relationships are already defined in the database, they just need to be accessed. The ADIMENS database also allows you to select or define groups of related data based on up to 150 distinguishing characteristics, with information from more than one database used in the selection process.

Data Analysis

ADIMENS contains an integrated statistical analysis package for analysing data. Alternatively, data can be extracted from the database and analyzed programmatically.

ADIMENS allows you to analyze your data to suit your needs, by allowing you to define (and save) your own equations and operations. This is extremely helpful when you need to go beyond the normal analytical abilities of most statistical analysis packages. These operations that you define can be saved, recalled, and re-used any number of times. Remember again that these operations can be used from directly within the database, or programmatically.

Data Presentation

After you have collected your data, manipulated it, and analyzed it, it is helpful (and often necessary) to present this information in the form of a report, or perhaps graphically. ADIMENS makes it easy for you to do both. The integrated word processing package, TEDI, allows you to access information from the database, and integrate it into forms, letters or reports. This not only saves time, but it eliminates the human errors that occur when information is manually transported from one source to another.

If you have a very specialized format for presenting data, a simple application program (written in FORTRAN, C or Pascal) can be used to extract the necessary information from the database, and present it in the desired manner. And this same application program can be used to extract and present data from any ADIMENS database file, on any ADIMENS-compatible machine.

PC-CAD BY ITSELF IS A GIANT STEP BACKWARD

Hector Holguin, P.E.
The Holguin Corporation
5822 Cromo Drive
El Paso, Texas 79912

INTRODUCTION

In the real world of CAD, you must develop a plan with a strategic focus to achieve your ultimate automation goals. Architects and engineers by the thousands are moving to PC products to fulfill their CAD automation goals. Unfortunately, valuable time and resources will be wasted as they unravel the proper application and direction of their new tools. The ever-accelerating pace of technology only magnifies this problem. The PC of today does not have the controls or horsepower to govern the massive boundaries of design and drafting operations. The HP Vectra by itself is a giant step backward because it ignores the automation advantage of the more Advanced HP workstations (HP 9000/1000 series). Major improvements in productivity must be realized to achieve true automation; improved drafting quality by itself is not enough. You must plan and manage the design and drafting process with the best net-working combination of HP Vectra and Advanced HP workstations.

LOST OPPORTUNITIES

Even the smallest of firms might appear secure in their PC investment, but the corresponding confined scope of operations and lost opportunities will severely limit their positioning for the future. YOUR BEST PEOPLE MUST HAVE THE BEST TOOLS TO CHALLENGE THE LEADING EDGE OF CAD BENEFITS.

A NEW GENERATION

A new generation of master craftsmen must have the proper tools to produce the designs of tomorrow. Traditionally, architects and engineers demand the highest standards for creative, quality, functional and cost-effective production. In our rapidly changing world, we must optimize the skills, experience and creativity of each designer and drafter.

PRIMARY CONSIDERATIONS

Each typical drawing file, architectural or engineering, demands countless computations and large storage requirements with precise organization and instant manipulation of each element of data. In even the smallest of operations, thousands of elements must be constantly massaged (precise mathematical relationships, copies, transfers,

compilations, derivatives, etc.). Each CAD operation will ultimately demand instant storage and access to hundreds of drawings and thousands of standards (details, symbols, notes, templates, patterns, etc.). Typically, within the first two years of operation, the investment in the CAD library of standards will exceed the cost of the hardware and software.

LEVELS OF AUTOMATION

IT IS CRITICAL THAT YOU RECOGNIZE THAT THERE ARE DIFFERENT LEVELS OF AUTOMATION. And, the HP Vectra will only serve the first level of drafting, design and analysis. The HP Vectra is too limited and too costly beyond the introductory level of CAD applications. You must analyze the memory and hard disc requirements to produce, store and access a typical drawing file. You must observe the interactive processing speed and production benchmarks of the Advanced HP workstations. You must dissect the operating system and disc management tools of these advanced systems. The HP 9000 and HP 1000 series support the proper internal controls for sophisticated data manipulation, editing, filing, backups, background plotting, etc.

A BALANCED NETWORK

YOUR AUTOMATION STRATEGY MUST INCORPORATE A PROPER BALANCE OF HP VECTRA AND ADVANCED HP WORKSTATIONS. You must become acutely aware of the severe limitations of a CAD investment that does not have a proper balance of HP Vectra and Advanced HP workstations. A balanced investment will give you the automation advantage in the evolving high-tech world of design and drafting. A CAD network can include HP Vectras but only with the proper alignment and integration to the more Advanced HP workstations. The productivity gain from each HP Vectra plus its ability to support a broad range of PC software must contribute to a cost-effective network and a dynamic growth path to Advanced HP workstations for your more creative and productive people.

THE GOVERNING FACTOR

Once you amortize the cost of a CAD system over a 5 year period, the hourly cost of the operator (salary and benefits) becomes the governing factor. The following simplified example will highlight the importance of moving your best people to a higher level of automation. This example is based on a 5-year amortization of the workstation cost; the Advanced HP workstation is assumed to outperform the HP Vectra by a productivity gain of "2-to-1" (at least 2-to-3 times faster based on conservative industry reports.... advanced functions, operations and controls multiply the productivity potential).

COST OF WORKSTATION

\$10,000
30,000

COST PER HOUR

\$1.00
3.00

1. Assume HP Vectra cost to be \$20 per hour
\$1 per hour PC cost plus \$19 per hour operator cost
(operator cost includes salary and benefits)
2. Assume Advanced HP workstation cost to be \$22 per hour
\$3 per hour station cost plus \$19 per hour operator cost
3. Assume one-month project using HP Vectra
Total cost = \$20/hour X 173 hours = \$3,460
4. Assume Advanced HP station reduces time to 86.5 hours
Total cost = \$22/hour X 86.5 hours = \$1,903
5. Total savings per workstation per month
 $\$3,460 - \$1,903 = \$1,557$
6. Total savings per workstation per year
 $\$1,557 \times 12 = \$18,684$
7. Total system savings per year
Multiply \$18,684 by number of expected workstations

In this example, the difference in cost between an HP Vectra and an Advanced HP workstation can be recovered within the first year of operation.

COST-EFFECTIVENESS

Since all companies are in the business to make a profit, it is important to determine the cost-effectiveness of all capital investments. A CAD system is a capital investment. Too often companies who purchase CAD systems only consider the increase in productivity without carefully analyzing the cost-effectiveness of the acquisition. Productivity is one factor among many within the scope of evaluating cost-effectiveness. Cost-effectiveness parameters provide a more comprehensive evaluation process.

COST-EFFECTIVENESS relates costs of CAD methods to manual methods, whereas productivity relates the speed of CAD methods to manual methods.

COST-EFFECTIVENESS PARAMETERS

The following parameters affect cost-effectiveness and, therefore, should be considered in evaluating various CAD alternatives:

1. Purchase Price of System

The total cost of all CAD workstations (software, hardware, training, maintenance, support, etc.).

2. Workstation Cost

The allocated cost of each workstation is determined by dividing the total purchase price of the system by the number of workstations. It is important to calculate the single workstation cost; the cost-effectiveness analysis must compare CAD methods to manual methods for the same unit of output.

3. Productivity or Productivity Ratio

Ratio comparing the speed of producing a drawing with a CAD operation relative to a manual operation. Productivity evaluations must consider the following factors:

- a. Management of system
- b. Nature of work
- c. Software characteristics
- d. Hardware characteristics
- e. Training period - The amount of time that it takes to learn the CAD operations in order to achieve a productivity level equal to or greater than manual methods. With all systems, the user will experience a period where manual methods will be faster than CAD methods to produce a drawing. Some systems will enable the user to achieve a 1-to-1 productivity in one week of operation, whereas others may take six months or more. The training period has an important impact on the cost-effectiveness and payback period of each CAD system.

4. Operator Cost

Operator cost includes salary and benefits (vacation, sick leave, insurance, profit sharing, overhead allocation, etc.).

5. Overhead Cost

The cost associated with changing any facilities to accommodate a CAD system. Some systems require extensive changes, others require no changes, and others may even reduce the overhead costs.

6. Maintenance of System

The cost of maintaining the hardware and the software. Some companies include software updates in their maintenance price; therefore, you may not want to view the total maintenance payment as an expense for analysis purposes.

7. Other Factors

Other items that must be defined for a proper evaluation process:

- a. Interest cost of money
- b. Period of amortization for capital assets
- c. Present payroll drafting costs, including payroll overhead such as medical insurance, pension plan, FICA, etc.

Using the above factors, the formula derived below will produce the following results:

1. Compare the cost-effectiveness of any CAD system.
2. Compare the cost of manual methods to CAD methods and cost-justify the benefits of automation.

COST-EFFECTIVENESS FORMULA

LET:

1. CE = Cost-Effectiveness
2. WS = Workstation Cost
3. MWS = Monthly Workstation Cost, which includes interest, maintenance, etc.
4. P = Productivity Factor
5. MOPC = Monthly Operator Cost for CAD
6. MOH = Monthly Overhead Cost for CAD
7. M = Maintenance of System (expressed as percentage of WS)
8. MDC = Monthly Drafter's Cost (Payroll plus Payroll Overhead) for manual methods
9. AP = Amortization Period for Capital Expenditures
10. I = Interest Rate on Money

COST-EFFECTIVE FORMULA

$$\text{Cost-Effectiveness} = \frac{\text{MDC}}{\frac{\text{MOPC} + \text{MWS} + \text{MOH}}{P}} \quad \text{This provides a ratio of manual drafting cost to CAD cost}$$

Supporting Formulas:

1. Monthly Cost of Drafting with CAD = MOPC + MWS + MOH

2. Monthly Cost of Drafting
Using CAD for same output as

$$\text{Manual Methods} = \frac{\text{MOPC} + \text{MWS} + \text{MOH}}{P}$$

$$3. \quad \text{MWS} = \frac{\text{WS}[1 + (I \cdot \text{AP})]}{\text{AP}} + \frac{\text{WS}(M \cdot \text{AP})}{\text{AP}}$$

$$4. \quad \text{MOH} = \frac{\text{OH}[1 + (I \cdot \text{AP})]}{\text{AP}}$$

COST-EFFECTIVENESS EXAMPLE

Question: What are the savings of producing a drawing with CAD methods over manual methods?

Assumptions: Evaluate the following two CAD workstations:

WS1 = HP Vectra
WS2 = HP 9000 Model 320

The purchase price of the first workstation, WS1 = \$10,000; the purchase price of the second workstation, WS2 = \$30,000.

Workstation Cost WS1 = \$10,000 WS2 = \$30,000

Monthly Drafter Cost MDC = \$2,600

Productivity P1 = 1.5-to-1; P2 = 3-to-1

Based on Industry benchmarks, the productivity rating for HP Vectra is set at 1.5-to-1 and the HP 9000 Model 320 is set at 3-to-1.

Monthly Operator Cost MOPC = \$2,600 (same as MDC)

Overhead OH = 0

Maintenance M = 10% of WS per year (.833% per month)

Interest I = 13% per year (1.083% per month)

Amortization Period AP = 60 months

First compute MWS (Monthly Workstation Cost) as follows:

$$\begin{aligned} \text{MWS} &= \frac{\text{WS}[1+(I*AP)]}{AP} + \frac{\text{WS}(M*AP)}{AP} \\ \text{MWS1} &= \frac{10,000[1+(.01083*60)]}{60} + \frac{10,000(.00833*60)}{60} \\ &= 275 + 83 \\ &= \underline{\$358} \\ \text{MWS2} &= \frac{30,000[1+(.01083*60)]}{60} + \frac{30,000(.00833*60)}{60} \\ &= 825 + 250 \\ &= \underline{\$1,075} \end{aligned}$$

Cost-effectiveness is computed as follows:

$$CE = \frac{MCD}{\frac{MOPC + MWS + MOH}{P}}$$

$$CE1 = \frac{2600}{\frac{2600 + 385 + 0}{2}}$$

$$= \frac{2600}{1972}$$

$$= 1.32$$

$$CE2 = \frac{2600}{\frac{2600 + 1075 + 0}{2}}$$

$$= \frac{2600}{1225}$$

$$= 2.12$$

FIRST WORKSTATION (HP VECTRA)

For the same unit of output, it will cost you 1.32 times more with manual methods. Another way to interpret these results is to invert the cost-effective ratio from 2600/1972 to 1972/2600 which yields 76%. Thereby, the HP Vectra provides a 24% savings over a manual operation.

SECOND WORKSTATION (HP 9000 MODEL 320)

The HP 9000 Model 320 yields 47% (1225/2600) and a 53% savings over a manual operation.

COST-EFFECTIVENESS COMPARISONS WITH LEADING CAD SYSTEMS

This study will assume a balanced network of four HP workstations (2 Model 320 and 2 HP Vectra). These four workstations are sharing all peripherals and the prorated cost for each workstation is \$25,000.

Based on the typical cost and performance benchmarks of the leading CAD systems, the tables in this section will evaluate the cost-effectiveness of the corresponding range of CAD workstations. These benchmarks are compared to the time and cost of producing the same drawing on a DRAFTING TABLE.

IS THE DRAFTING TABLE OBSOLETE?

Purchase Cost Per Workstation WS	Monthly Cost Per Workstation MWS	Monthly Cost of Operator MOPC	Total Monthly Cost D	Equivalent Monthly Cost *2 to 1 E	Equivalent Monthly Cost *3 to 1 F
\$ 25,000(HP)	\$ 896	\$2,600	\$3,496	\$1,748	\$1,165
50,000	1,792	3,120	4,912	2,456	1,637
75,000	2,687	3,120	5,807	2,904	1,936
100,000	3,583	3,120	6,703	3,352	2,234
150,000	5,375	3,120	8,495	4,248	2,832

*2-to-1: One person on CAD workstation produces work of two persons on drafting tables.

*3-to-1: One person on CAD workstation produces work of three persons on drafting tables.

WS: The first column above, "WS", is the total cost of a typical four workstation CAD system divided by 4 to establish the purchase cost distribution per typical workstation. Each system must be totally operational.....disc storage, tape archiving, plotter, digitizer, software, etc.

MWS: The second column above, "MWS", is the monthly cost per workstation based on 5 year amortization, 13% annual interest, 10% annual maintenance and software support.

$$MWS = \frac{WS[1+(.01083 \times 60)]}{60} + \frac{WS(.00833 \times 60)}{60}$$

$$= 0.275WS + .00833WS$$

$$= .03583WS$$

MOPC: The third column above, "MOPC", is the monthly cost of operator for each workstation. Assume that the \$25,000 HP workstation uses existing staff without requiring specialist training; thereby, the identical cost of a person on a drafting table is assigned to this workstation.

Assume \$15 per hour (includes salary, benefits, overhead)

$$MOPC = \frac{15 \times 2080}{12} = \$2,600 \text{ per month}$$

Most systems require specialist training and generally provide increased salary and benefits. Assume \$18 per hour (includes salary, benefits, overhead)

$$\text{MOPC} = \frac{18 \times 2080}{12} = \$3,120 \text{ per month.}$$

D: Column "D" above is the total monthly cost of workstation plus operator.

$$D = \text{MWS} + \text{MOPC}$$

E: Column "E" above is the equivalent monthly cost of producing the same drawing on a CAD workstation based on a production multiplier of 2-to-1. One person on CAD workstation produces equivalent work of two persons on drafting tables.

$$E = D/2$$

F: Column "F" above is identical to E, but based on production multiplier of 3-to-1. One person on CAD workstation produces equivalent work of three persons on drafting tables.

$$F = D/3$$

IMPORTANT: THE TYPICAL PRODUCTIVITY GAIN BEING REPORTED BY MOST FIRMS/ AGENCIES IS IN THE RANGE OF 2-TO-1 TO 3-TO-1. Thereby, this study will not exceed this range to establish the cost-effective benefits of each workstation.

COST-EFFECTIVE RATING		
Purchase Cost per Workstation WS	2-to-1 G	3-to-1 H
\$ 25,000	-33% Savings	-55% Savings
50,000	- 6% Savings	-37% Savings
75,000	+12% Extra Cost	-26% Savings
100,000	+29% Extra Cost	-14% Savings
150,000	+63% Extra Cost	+ 9% Extra Cost

**NOTE: This table does not include training costs.

G: Column "G" above compares the cost of producing the same drawing on a CAD workstation and a drafting table, based on production multiplier of 2-to-1.

$$G = \frac{(E-2600)100}{2600}$$

$$G = \frac{E}{26} - 100$$

H: Column "H" above is identical to "G", but based on 3-to-1 production multiplier.

$$H = \frac{F}{26} - 100$$

NOTE: For both "G" and "H".

Negative (-)% = Cost to produce drawing on CAD workstation is less than cost to produce said drawing on drafting table.....SAVINGS.

Positive (+)% = Cost to produce drawing on CAD workstation is more than cost to produce said drawing on drafting table.....EXTRA COST.

Purchase Cost Per Workstation WS	Training Cost Per Workstation I	Yearly \$ Savings or Increase 2-to-1 J	Yearly \$ Savings or Increase 3-to-1 K
\$ 25,000	\$ 1,750	-\$40,896	-\$68,880
50,000	29,472	- 6,912	- 46,224
75,000	34,842	+ 14,592	- 31,872
100,000	40,218	+ 36,096	- 17,568
150,000	50,970	+ 79,104	+ 11,136

I: Column "I" above is the total training cost per workstation.

Assume that the \$25,000 HP workstation requires a maximum learning cycle of one week. Each operator is fully trained within 5 days with increased productivity and profitability in the second week of operations.

Assume 1/2 month of training:

$$I1 = \frac{MWS+MOPC}{2} = \frac{896 + 2600}{2} = \$1,748$$

use \$1,750

Most systems require months of extensive specialist training. Increased productivity of 2-to-1 does not typically occur until the second year of operations.

Assume 6 months of training:

$$\begin{aligned} I2 &= 6 (MWS+MOPC) \\ &= 6 MWS+(6 \times \$3,120) \\ &= 6 MWS+\$18,720 \end{aligned}$$

CAUTION: Six months may be too low. Based on today's CAD installations, many firms are experiencing training costs up to 50% of the initial system costs. The training costs above do not include the supervisory (management) time required in the training process.

J: Column "J" above is the yearly savings or increase in production costs by each CAD system (all 4-workstation). Based on 2-to-1 production multiplier.

$$J = (12 \times E \times 4) - (\$31,200 \times 4)$$

4 workstations Annual cost of drafting table

$$J = 48E - \$124,800$$

Negative (-) \$ value represents SAVINGS

Positive (+) \$ value represents INCREASED COSTS

K: Column "K" above is identical to "J", but based on 3-to-1 production multiplier

$$K = 48F - \$124,800$$

NOTE: "J" and "K" values do not account for the training costs; these costs will be deducted below.

ACCUMULATED SAVINGS					
Total System Cost L	Year 1 M	Year 2 N	Year 3 P	Year 4 Q	Year 5 R
\$100,000	-39,146	-108,026	-176,906	-245,786	-314,666
200,000	+29,472	+ 22,560	- 23,664	- 69,888	-116,112
300,000	+34,842	+ 49,434	+ 17,562	- 14,310	- 46,182
400,000	+40,218	+ 76,314	+ 58,746	+ 41,178	+ 23,610
600,000	+50,970	+130,074	+141,210	+152,346	+163,482

L: Column "L" above is the total cost of each 4-workstation system.

$$L = 4 \times WS$$

M: Column "M" above is the SAVINGS or EXTRA COSTS incurred in the first year of CAD operations.

Assume the \$25,000 HP workstation achieves 2-to-1 production multiplier within a few weeks.

$$M1 = J + I$$

$$M1 = -\$40,896 + \$1,750$$

$$M1 = -\$39,146 \text{ SAVINGS}$$

Other systems do not achieve 2-to-1 production multiplier until second year. Even though other costs are incurred, this column will only be assigned the training costs of column I.

$$M2 = I$$

N: Column "N" above is the ACCUMULATED SAVINGS or EXTRA COSTS incurred in the first two years of CAD operations.

Assume the \$25,000 HP workstation achieves 3-to-1 production multiplier in second year.

$$N1 = M + K$$

$$N1 = -39,146 - 68,880$$

$$= -\$108,026 \text{ SAVINGS}$$

Assume other systems achieve 2-to-1 production multiplier in second year.

$$N2 = M + J$$

P: Column "P" above is the ACCUMULATED SAVINGS or EXTRA COSTS incurred in the first three years of CAD operations.

Assume all systems at 3-to-1 production multiplier by third year.

$$P = N + K$$

Q: Column "Q" above is identical to "P", but based on accumulating first four years of CAD operations.

$$Q = P + K$$

R: Column "R" above is identical to "P", but based on accumulating first five years of CAD operations.

$$R = Q + K$$

CONCLUSIONS

1. The \$25,000 HP workstation has a complete cash payback within the first two years of operation.
2. Based on accumulated savings over a drafting table, none of the other systems produce a cash payback within the first five years of operations. The closest competitive 4-workstation system (\$200,000) has only accumulated savings of \$116,112.
3. WHAT PERCENTAGE OF THE TOTAL DESIGN AND DRAFTING WORK is the tedious placement of lines on paper, standard details, patterns, symbols, notes, title blocks, repetitive elements, derivatives of standards, mathematical models, etc.? The replacement of the DRAFTING TABLE is where the maximum benefits of CAD can be established.
4. COST-EFFECTIVE WORKSTATIONS will force the leading CAD vendors to reevaluate their expensive workstations for drafting applications, especially if the cost-effective workstations can support I.G.E.S. and its electronic transfer of drawing files to other CAD systems for specialized design functions.

RECOMMENDATIONS

Expensive workstations must be eliminated from the drafting functions. The cost of these systems can be dramatically reduced by tailoring the

expensive workstations to specialized design and analysis applications that can justify these expenditures. Each CAD operation must create a NETWORK that can properly support (by cost and performance) their future needs. Clusters of cost-effective workstations can readily feed production drawing files and mathematical models to specialized design workstations. A COST-EFFECTIVE NETWORK must govern each CAD decision.

THE LEADING EDGE

Each design and drafting operation must keep pace with the latest technology. Each time that they are ready to expand their CAD operations, they simply move their more productive people to the latest and most Advanced HP workstations available and network this capability to their current operation. The power and dynamic opportunities that they will gain by moving through the total world of CAD is true design automation. They must plan and manage the automation process. They must not just throw hardware and software at the problem or they will tend to mechanize and not automate. The only alternative is to reach for the leading edge of CAD technology and benefits.

The Design of a Graphical Database for the DRAWIT Drawing System

**Marc Katz
Graphicus
160 Saratoga Ave #32
Santa Clara, CA 95051**

OVERVIEW

The Drawit database is a graphics database capable of storing graphics data and accessing it for display, picking, archival and for user manipulation. The database was designed for the Drawit drawing system, which is an interactive drawing system targeted for scientific, engineering and general drawing applications.

The Drawit product is composed of three parts: the user interface, the database and the device interface. The user interface interacts with the user through pop-up menus and text forms to determine what graphical primitives need to be created and manipulated. The user interface controls the database to manipulate, create and display graphical primitives.

The device interface is used to display graphics in a device independent manner. Text generation, blanking and clipping are handled at the device interface level. The device interface uses a graphics subsystem like CORE or GKS (DGL is used on the HP1000) to interface to devices.

GOALS

The database project had several, often conflicting goals. Minimizing code space was a major consideration because the Drawit product must run on E and F series computers, which are very space limited. Minimizing data storage space and access time were often in conflict. Also the code had to be portable since this product will eventually run on several different machines. The database is written in Pascal, using as few system dependencies as possible.

STRUCTURE

Extents

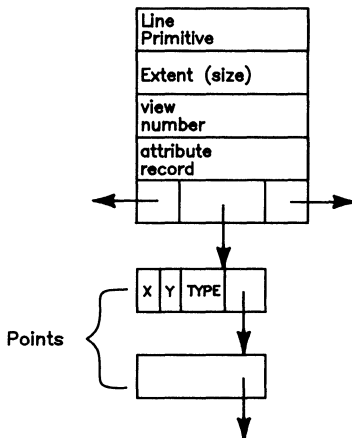
One of the most important design elements of the database is the use of extents. The extent of a primitive is the maximum area it covers at its North, South, East and West edges. Extents are used to optimize operations such as clipping and picking. Extents are maintained for text and line primitives and for groups. The extent of a group includes the extent of every item included in the group.

Lines

Line primitives are connected line sequences known in the graphics world as polylines. They may be quite large (hundreds of line segments) or small (two points for a straight line). Each line has an associated set of attributes, including color, linewidth and linestyle for its edge and fill lines, fill angle, density, hatching and a spline attribute.

Lines are stored as a sequence of points connected by pointers. Five words of storage space is required per x,y,opcode triplet. In addition, a two-word pointer is used to link them together. While it would have been more efficient to store lines in one contiguous area (thus avoiding those two-word pointers) it would have created other difficulties. Pascal does not support the ability to dynamically allocate arrays of variable size. Also, because we support the ability to insert and delete points the linked list structure is very convenient.

Line primitive structure



A line has a list of points in a singly linked list structure. The point record contains an x,y pair and a type indicating a move or draw.

There are only two primitives in the Drawit system: lines and text. This is different from many graphics systems that also store arcs, circles, ellipses and

others. While arcs and circles can be created, they are stored as connected line sequences like any other line. The advantage is that they may be scaled and rotated, displayed, retrieved, edited and archived like any other line. This saves code space and complexity.

We were able to optimize our treatment of lines because we had only two kinds of primitives to worry about. In this environment, data space is more available (through the virtual memory area) than code space. This is very similar to the tradeoffs inherent in a RISC architecture where a small number of optimized instructions are used.

One disadvantage of this scheme is that storage of circles and arcs uses more data space than it would if only the center and radius were stored. This disadvantage is partially alleviated by using the spline attribute with circles and arcs. Circles and arcs are generated with only enough points to make them reasonably smooth, thus minimizing the space used. The spline attribute can be used to make these primitives smoother.

Text

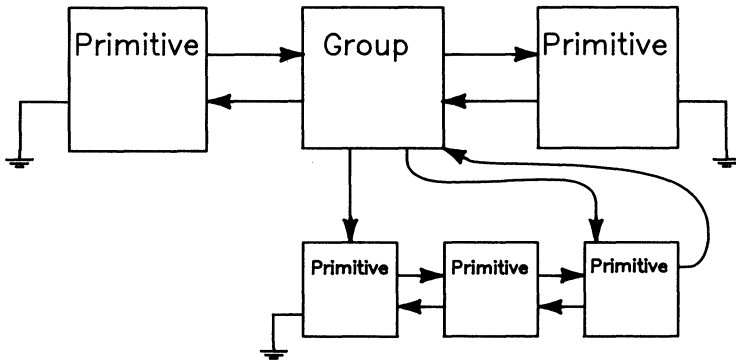
Text has 17 attributes affecting its appearance, including height, aspect ratio, gap, line spacing, font, slant and rotation. These attributes affect the size and position of the text. Text primitives may have a large number of strings that are displayed as a block of text.

When text is scaled or rotated, the attributes must change in response. Height, rotation, aspect ratio and text location may be affected by transformations. The database gets information about text transformations by interacting with the device interface.

Groups

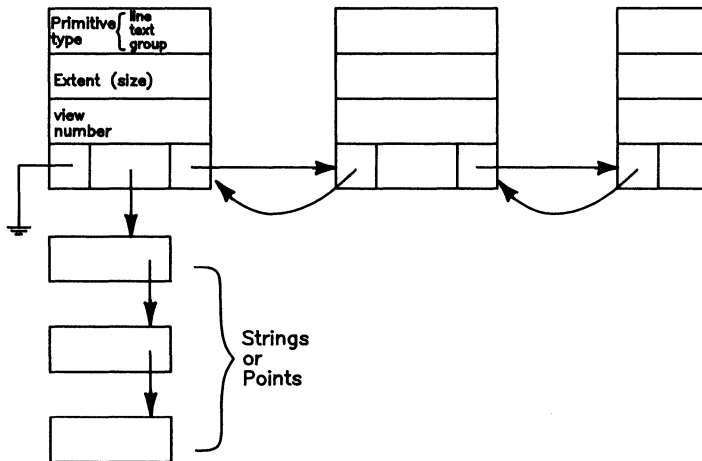
Groups allow several primitives to be treated together for purposes such as copying, transforming, etc. Actually, groups can be viewed as a third type of primitive item because in all operations, the group must be acted upon and its extent updated. The database has only one level of grouping (a group cannot contain other groups).

Structure of a Drawit group



This figure shows a group containing three primitives. Note that pointers are maintained to the first and last primitive in the group so the group can be traversed both forwards and backwards.

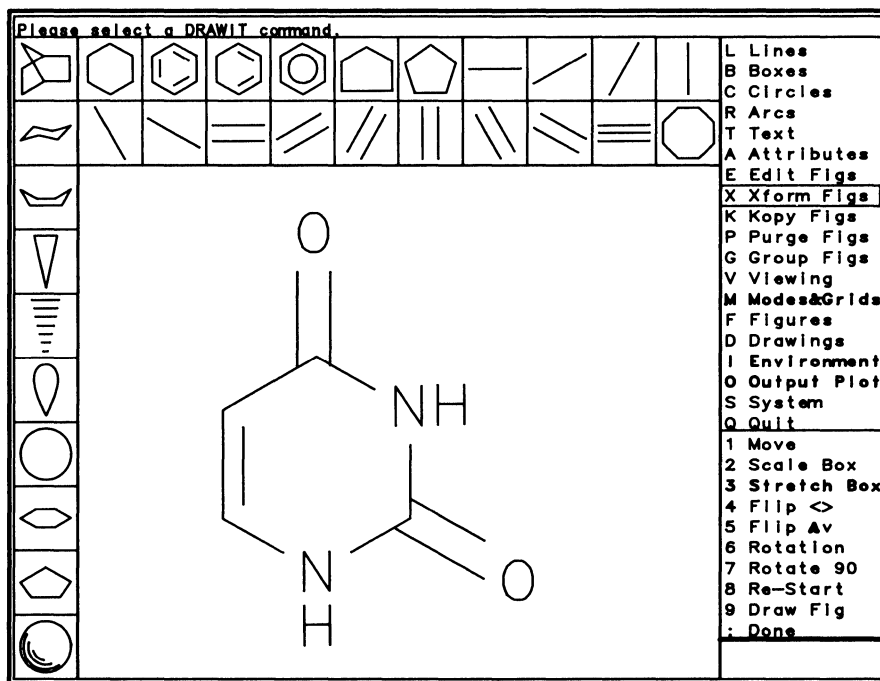
Drawit database structure



A list of objects may be defined for each viewing window. This list is traversed to redraw the window or to select an object.

Viewing

The database is a multi-window oriented system. While there is only one drawing area in Drawit, there are two figure menu areas that contain figures which can be placed onto the drawing and a prompt line. Also, the viewing system is used for the pop-up menu areas to allow the selection of menu items with the graphics cursor. The following figure shows a Drawit screen that shows five windows.



Each window has a viewing transformation associated with it and additional information about clipping and coloring. A window may also have an object list associated with it that stores primitives and groups to be displayed in that window. Zoom and Pan operations are performed by changing the viewing transformation. Changes in the location of the drawing or figure menu areas are also done that way.

By setting up a flexible windowing system, we made it possible to implement the basic drawing features, menu picking and the figure menu feature. We can also use this to add additional capabilities like layers, zoom windows, etc.

ALGORITHMS

Picking

Picking is the selection of a primitive or group with the graphics cursor. When the user inputs a point, a search is made to find a primitive which is close to the cursor (this region is called the pick aperture). What if several primitives are close to the cursor?

A common solution is to search the whole database, returning the primitive that is closest to the cursor. A great disadvantage of this scheme is that the entire database is always searched. Also, complexities arise when defining the meaning of "closest." Filled regions can be picked anywhere inside their boundary. If the cursor is placed near a line lying on a filled region, which is closest? Many pick algorithms have to make special rules for filled areas which give favor to lines intersecting them. If two primitives lie on top of each other, which is closer?

A better picking algorithm takes into account the true behavior of users. Users usually pick objects in a place where there is no confusion; therefore in the vast majority of cases, the first object in the pick aperture is the object desired. When picking in a dense area, it is not likely that the user would be able to determine closeness and use it to pick the correct object. In this case, the user must be allowed to cycle through possible selections until the correct one is obtained.

Our algorithm returns the first primitive within the pick aperture. In most cases this produces the correct selection because most users try to pick in an uncluttered area. We also provide a "next pick" function that allows another pick in the same location to result in the continuance of the search. This solution surmounts all complications in picking by giving the user control.

Clipping

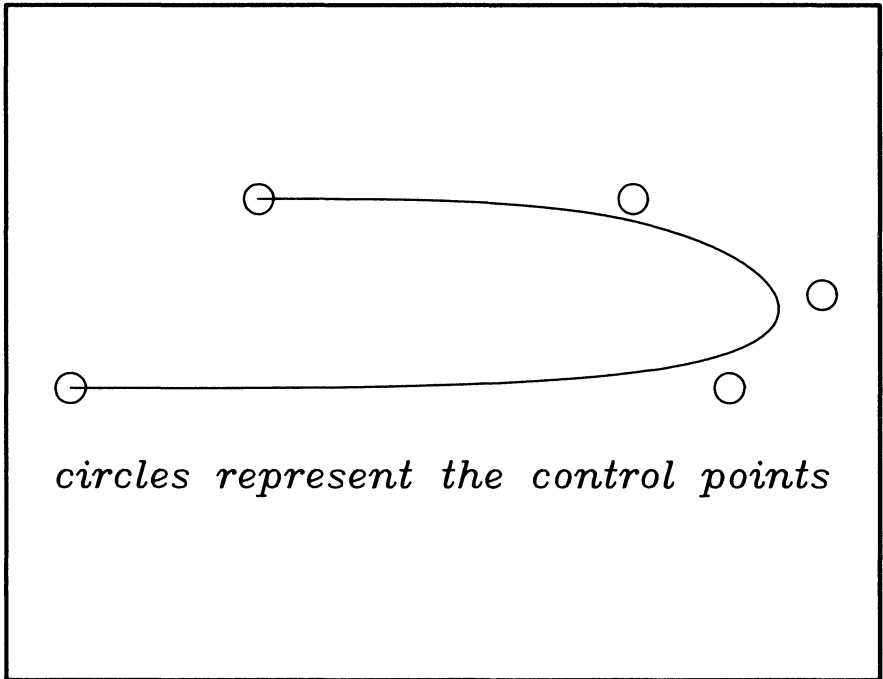
Clipping causes primitives that are not totally visible to be "clipped" at the drawing area boundary. Primitives that are partially on the display need to be intersected with the edge of display area so only visible sections are drawn. Although clipping is performed in the device interface level, checking is done in the database using extents to prevent unnecessary processing.

When clipping a primitive, if the primitive is completely outside the window it need not be clipped or drawn at all; if it is completely inside the window, it must be drawn but does not need to be clipped at all. Only if the extent is partially inside the window does clipping need to be performed. Similarly for picking, only objects whose extent includes the pick aperture need to be processed further. The expense of storing and maintaining the extent is amply paid back by increased drawing speed and picking response. These efficiencies are especially great when zoomed in since then many of the objects are not displayed and should not be considered at all in the clipping or picking algorithms.

B-spline smoothing

With traditional pen and paper drawing, curves can be drawn fluidly and easily. Many computer drawing systems are very limited in curve drawing, allowing arcs, circles and other simple curves only. Complex curves can be generated by piecing together simpler curves. This is not natural or easy.

Splines offer a good way to draw curves with a computer drawing system. A sequence of guiding control points are defined; a B-spline smoothing algorithm is used to draw a smooth curve controlled by those points. Spline is an attribute of lines; therefore any line may be smoothed by enabling the spline attribute.



Splines are not kept in the database because several different situations could alter the spline, requiring it to be regenerated. While the curve is being defined, it is very common to move or add control points to change the shape of the curve.

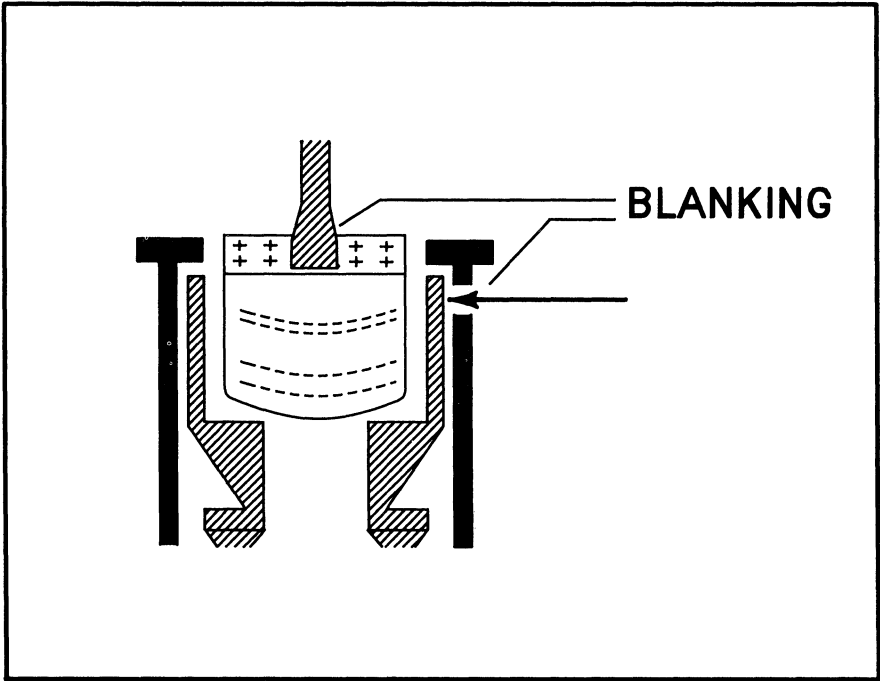
The resolution of the spline attribute is controllable for each device. On plotters, a high resolution spline is the default. Lower resolution is the default on terminals because less detail is needed. Fewer spline points may be mandated for filled regions since there is a limit on the number of points in a filled line. The spline

function will lower the resolution for lines that have too many points to be filled to make filling possible.

Splines must be picked also. Since they are not actually stored in the database but are generated for display, they must be generated for picking. To decrease the computational load, they are generated at very low resolution for picking. Also, they are not computed at all unless the pick aperture is within the extent of the line.

Blanking

Groups and text have a blanking attribute that allows them to be shielded from obstruction by lines. A common use would be placing a legend on a graph or placing text on top of a filled region. The following illustration shows some uses for blanking. Blanking is more commonly useful than we thought originally. It is very useful in complicated drawings and for special filling effects. Extents help decrease computation load here, too. The blanking algorithm need only consider objects with extents intersecting the blanked areas.



Transformations

Transformations are two-dimensional manipulations that can be applied to primitives or groups. The transformations we support are translation (move), scale, rotate and mirroring. Transformations can be implemented with a 3x2 matrix multiplication. This is not very efficient because the most common transformation is translation, which can be implemented by addition alone. Scale is the next most common and it requires an addition and a multiply. Only a complex transformation involving rotation plus scale or translation requires the whole 3x2 matrix multiply. By optimizing for translation and scale, these operations will use dramatically less CPU time.

CONCLUSION

The HP1000 computer environment is a demanding environment for large program development. It demands solutions that make the best tradeoffs between resource use and function. We have tried to show how computation could be minimized by optimizing for the most common cases and how the tradeoffs between memory use and performance were reconciled for what we hope is the best overall solution.

A General Purpose Process Graphics System

by: Walden, Phil

We regret that this paper
was not received for
inclusion in these proceedings.

The Design of GEDIT - A General Purpose Graphics Editor
Kurt Van Ness
Flexware Inc.
533 East "F" Street
Ontario, California 91764

1.0 Introduction

GEDIT is an interactive program used to create and modify drawing files, much like EDIT/1000 is used to create and modify text files. The basic input device is a block mode graphics terminal. Output devices range from plotters to dot matrix printers, laser printers, and film plotters.

2.0 Design Philosophy

The goal in designing GEDIT was to create a program with powerful functions that was also easy to learn and use. Functionality and ease of use are usually at odds with each other; simple programs, functions, commands are generally easier to use than their more complex counterparts.

To simplify the user interface, menus and forms are used. Because of all the functions GEDIT offers, there are a lot of menus and forms. The challenge comes in describing how all the menus and forms are used.

2.1 Demonstration and Online Help Capability

Initially, a reference manual was written with the purpose of describing each menu and function, and illustrating their use with examples. The problem with manuals is that users don't like reading them. In practice, few users ever actually read the GEDIT reference manual - the typical novice user learned by example from a more experienced user. The solution was to design an automated experienced user that teaches by example and can explain each operation as it is performed. The result was the development of a demonstration capability that allows tutorials and demos to be generated by experienced users and played back by novice users. In addition, an online help capability was developed which allows users to ask for help at any time.

3.0 Design Implementation

As with any graphics editor, GEDIT defines a set of objects and a set of operations that manipulate the objects. The objects, referred hereafter as entities, consist of; lines, arcs/circles, text, polygons, polygon fills, figures, and flowchart entities.

3.1 Entities

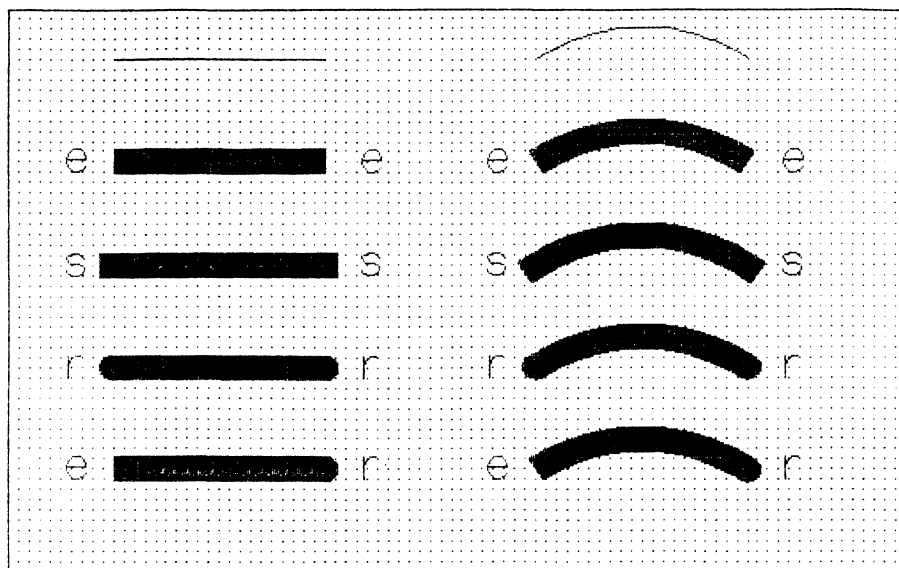
Each entity consists of properties, geometric or otherwise. Examples of non-geometric properties are layer and pen number. Because of limited space in this paper, only the more interesting properties will be described in any detail.

3.1.1 Lines and Arcs

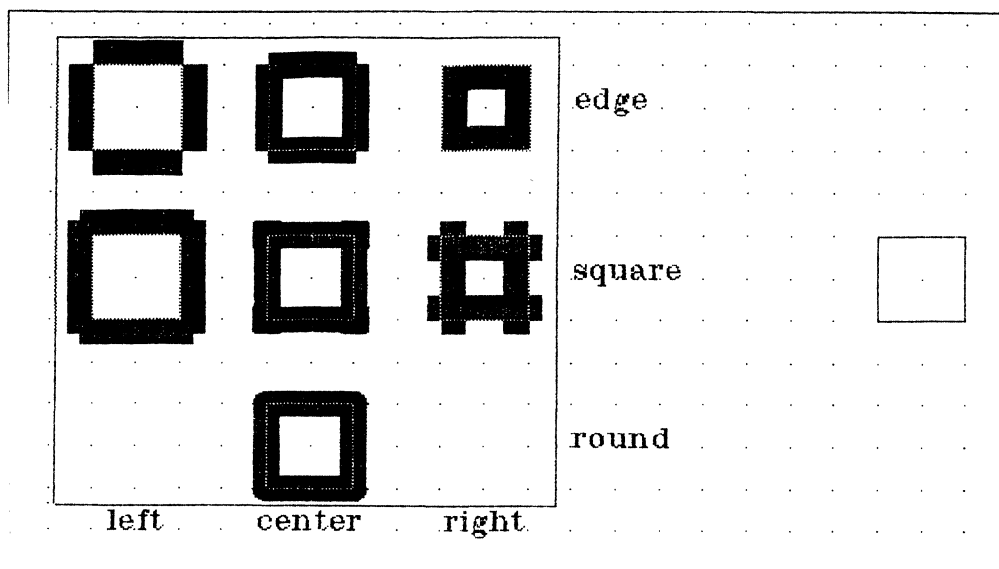
Geometrically, lines are defined by their end points, and arcs by their origin, radius, start and stop angles. Lines may be drawn by successively digitizing their end points. Digitizing is the process of positioning the graphics cursor and pressing a menu key, mouse button or tablet stylus. Lines may also be drawn by specifying one end point, an angle, and a length. Arcs may be specified as origin and radius with start and stop angles or three circumference points. Circles may be specified as; origin and radius, two circumference points, or origin and one circumference point.

3.1.1.1 Properties of Lines and Arcs

Some interesting properties of lines and arcs are width, justification, and end point profile. The width allows lines thicker than a pen to be drawn by making multiple pen strokes. The justification specifies how thick lines are justified relative to a "zero" width line. Justification may be to the right, centered or to the left. The end point profile is how the end points of thick lines are drawn. The profile may be edge, square, or rounded. The round profile is particularly useful when contiguous thick lines at odd angles are drawn since the gaps and voids at end points are eliminated.



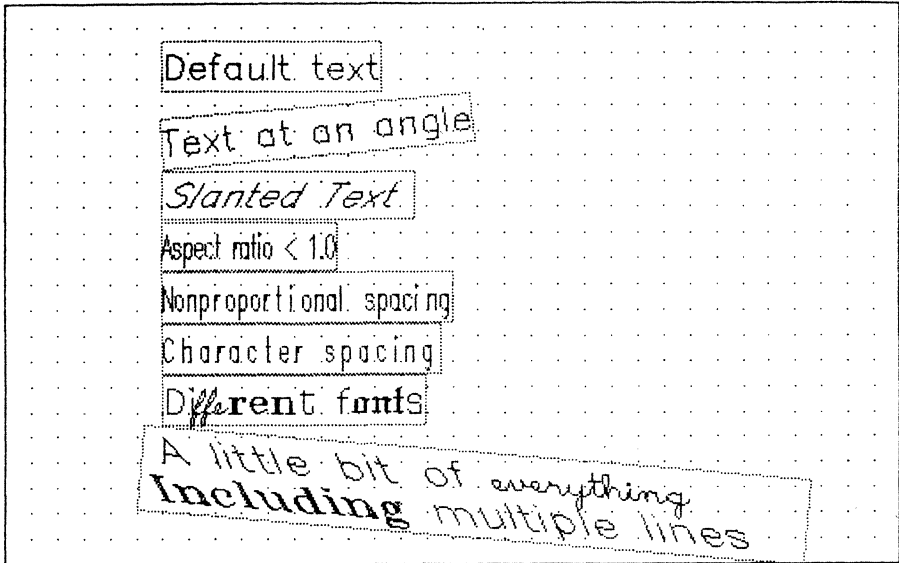
Line and Arc End Point Profiles.



Results of Justification with Different Profiles on Lines.

3.1.2 Text

Text is inserted as blocks. A block consists of 1 or several lines of text. Text has a rich set of attributes. Common attributes are height, angle, justification, font, and pen. The font and pen may be specified on a character by character basis. Other attributes are; layer, aspect ratio, slant angle, proportional/nonproportional spacing, character spacing, and line spacing.



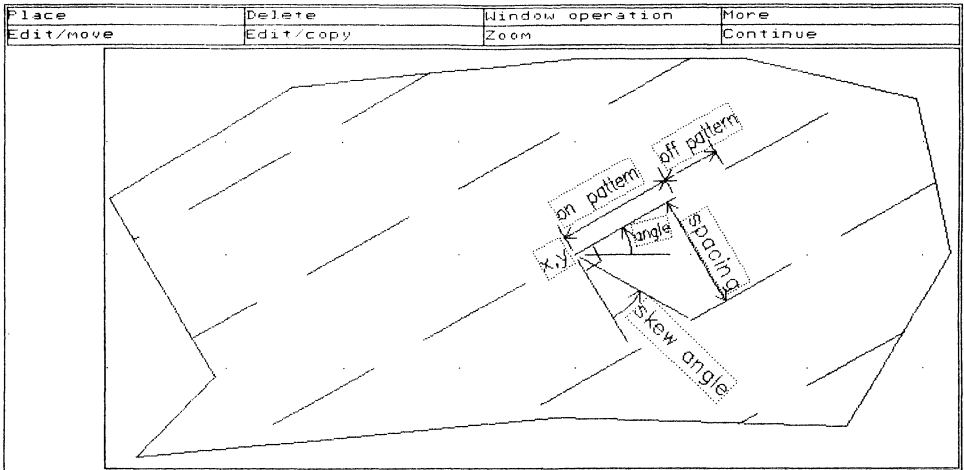
Text parameter examples.

3.1.3 Polygons

Polygons enclose an area. Polygons are composed of linear or circular edges (segments and arcs). Polygons may be constructed from existing lines and arcs or by digitizing vertex points. An automatic selection algorithm allows an entire polygon to be constructed by selecting a single edge.

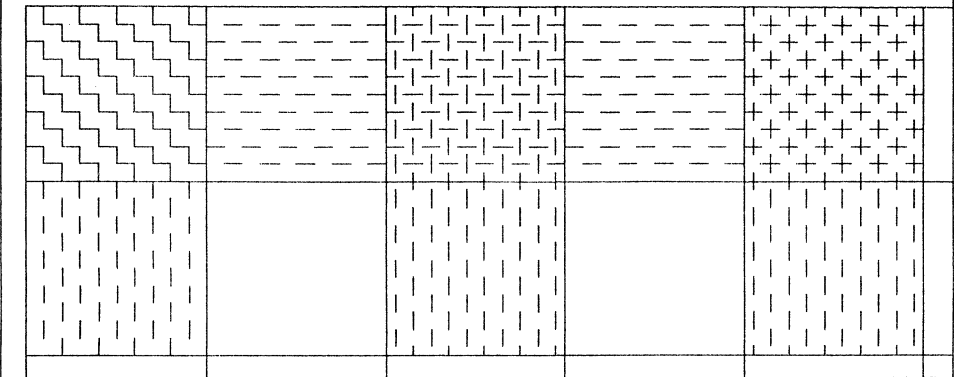
3.1.4 Polygon Area Filling

Polygon filling is accomplished by selecting one or more polygons and specifying a fill pattern. Area attributes are layer, pen, angle, pattern on length, pattern off length, skew angle, and origin.

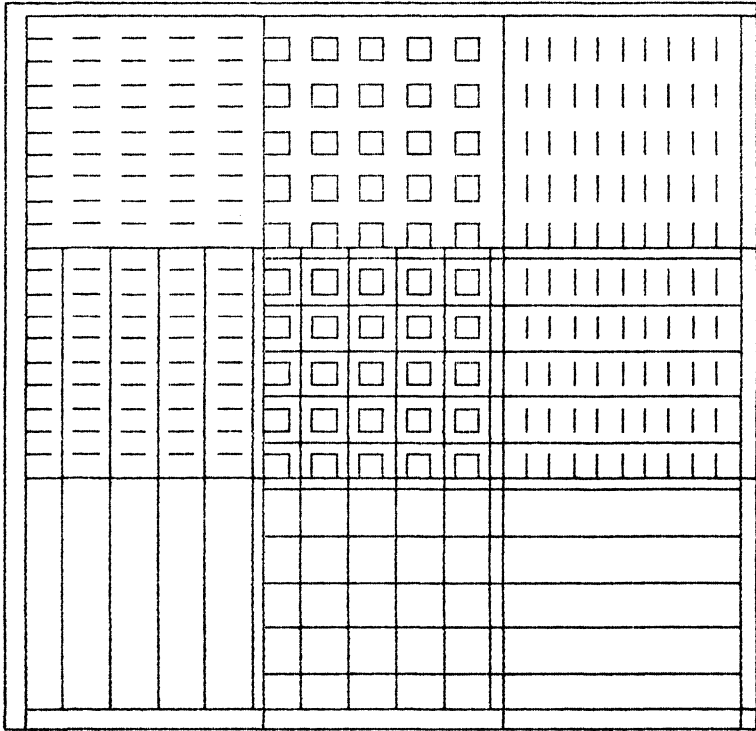


Basic polygon fill pattern.

Particularly unique attributes of GEDIT's fill pattern are the pattern origin, x and y, and the skew angle. The "on" portion of the fill pattern begins at the origin. The skew angle defines the alignment of the "on" portions of adjacent fill lines. Taking advantage of these attributes allows a large variety of interesting fill patterns to be generated by simply combining different combinations of the basic fill pattern.

Place	Delete	Window operation	More
Edit/move	Edit/copy	Zoom	Continue
			

As shown just above, a horizontally oriented, filled rectangle is combined with 3 vertically oriented, filled rectangles. The only difference between the 3 vertical filled rectangles is the location of the pattern origin - yet 3 entirely different fill patterns are generated in the overlapped areas.



As shown above, four different polygon fills are overlapped. Each polygon is a square 2 units on a side. The unit square in the middle is overlapped by all four polygon fills.

3.1.5 Figures

Figures are composed of lines, arcs, text, polygons, polygon fills, and other figures. They are analogous to a subroutine with respect to a program.

Figures are created by copying the desired entities from a drawing to a figure file. Once created, a figure may then be inserted into drawings. Attributes of figures are; origin, layer, scale, angle, and mirror/no-mirror.

3.1.6 Flow Chart Entities

GEDIT contains a special purpose interactive flow chart capability that allows flow charts to be drawn quickly and easily. When drawing flowcharts, several design rules are enforced. These rules prevent flow chart boxes from overlapping and don't allow flow chart lines to cut thru flow chart boxes. In addition, flowchart lines must be orthogonal - arrows may appear only at the end points of flow

chart lines. Jumps may appear at four way intersections of flow chart lines.

3.2 Drawing Modes

Several drawing modes are selectable. The most important among these are the units and grid definition. The basic unit of measure may be defined as an English or Metric unit. Examples are; inches, deci-inches, centimeters or millimeters. A displayed grid and a snap grid are definable. Grids assist in the accurate placement and alignment of entities.

3.3 Operations

Operations may be divided into several catagorizes; individual operations, binary operations, and window operations.

3.3.1 Individual Operations

The individual operation catagory consists of the move, copy, edit, and delete operations. An individual operation begins with selecting the entity to be operated on. This is accomplished by either of two methods.

3.3.1.1 Form Editing Method

Using the first method, the cursor is placed near the entity and a selection key from the menu is pressed. The attributes of the selected entity are then displayed in a form allowing the user to modify any of the entity attributes. Next another menu key is pressed that replaces or copies the entity with the new attributes. A line, for example, could have its width increased or be moved by changing its end point coordinates.

3.3.1.2 Source and Destination Marker Method

The second method is generally used for moving and copying entities. Two menu keys allow a "source marker" and a "destination marker" to be defined. Placing the cursor near an entity and pressing the "set source" menu key defines the position of the source marker and selects the entity. Moving the cursor and pressing the "set destination" menu key defines the destination marker position. Once the source and destination markers have been defined and an entity selected, the "move" and "copy" menu keys may be pressed resulting in a "move" or "copy" operation.

3.3.1.2.1 Move operation

If a "move" operation was specified, the selected entity is moved from the source position to the destination position.

Next, the source and destination marker positions are swapped; the moved entity remains selected. This allows the entity to be easily returned to its original position by simply pressing the "move" menu key once more.

3.3.1.2.2 Copy operation

If a "copy" operation was specified, the selected entity is copied from the source position to the destination position. Next, the source and destination markers are moved by the same displacement as the copied entity and the copied entity becomes the selected entity. This allows step and repeat operations to be performed by repeatedly pressing the "copy" menu key.

3.3.2 Window Operations

Window operations are so named because a window is used to select the entities to be operated upon. A simple window is defined by digitizing only two points. The points define the diagonal of an orthogonal rectangular area. A polygon window is defined by digitizing more than two points. Each point defines a vertex of the polygon.

Entities are selected on the bases of being inside, outside, or crossing the window or any combination thereof. Two powerful classes of operations may be performed on entities so selected: transformations and attribute editing.

3.3.2.1 Transformations

Transformation operators consist of displacement (move), rotation, scaling, and mirroring. For displacement, a source and destination point are digitized and the selected objects are moved from the source position to the destination position. Mirroring requires two mirror points to be digitized. These points define an axis about which the objects are mirrored. For rotation and scaling, a rotation point and a scaling point are digitized and the selected objects rotated or scaled about the respective points.

A stretch modifier allows lines, polygon edges, and polygon fill edges to be stretched if they cross the window. A copy parameter generates N copies of the selected entities by performing the transformation N times. If a delete option is specified, the selected entities are deleted instead of transformed.

3.3.2.2 Attribute Editing

Attribute editing allows the attributes of all the selected entities to be modified in a single step. Examples would be to change all the entity pen numbers to 4, increment widths by .1, or multiply layers by 2. For each attribute to be

modified, an operand value and an optional operator are specified. The operators may be addition, subtraction, multiplication, division, and modulo. The old attribute value is modified by the operator and the operand value. If an operator is not specified, the old value is simply replaced by the new value.

3.3.3 Binary Operations - Trimming and Dissection

Binary operations are so called because two entities are involved. The entities must be either arcs or lines. The "Select trim entity" menu key selects the trim entity. The "Trim" menu key selects another entity and trims it to the trim entity by extending or shortening the selected entity until it intersects the trim entity. The "Dissect" menu key splits the selected entity into two pieces at its intersection with the trim entity.

3.3.4 View Control - Zoom

The zoom function allows the user to select the size of the work window and to display any portion of the drawing in the work window. The work window is the rectangular area containing the current view of the drawing. Zooming allows portions of the drawing to be enlarged, if, for example, detailed work needs to be done in a particular area. It also allows the entire drawing to be redisplayed.

3.3.4.1 The Zoom Window

The zoom window is a small rectangular area representing the boundary of the drawing and is displayed on the screen while in the zoom menu. The user selects the new work window by digitizing the diagonal of a rectangle. Digitizations may be made in either the current work window or the zoom window. The zoom window is a key feature in allowing access to any portion of the drawing when "panning" or "zooming out".

3.4 Plotting

Plots can be generated on any device supported by HP's Device independent Graphics Library (DGL). If DGL is unavailable, plots can be generated on devices that understand HPGL (Hewlett Packard Graphics Language).

3.4.1 Scaling and Rotation

A plot scale may be specified during plotting which allows the drawing to be plotted the desired size. Also, the plot may be rotated 90 degrees before being plotted.

3.4.2 Pens

Plotting pens have the following attributes; width, speed, acceleration, and force. The values of each attribute may be specified individually for each pen. Varying the speed, acceleration, and force allows users to make their own quality versus speed tradeoffs. Specifying the pen width allows the polygon fill and thick line/arc generating routines to calculate the exact number of pen passes needed to fill an area. Alternatively, the user can cause any desired overlap or underlap during solid filling.

4.0 User Support Tools

4.1 GEDHELPGEN - On Line Help Generator

The GEDHELPGEN utility program allows the on-line help file, provided as part of the software package, to be modified and customized by the system manager.

4.2 GEDDEMOGEN - Demo Generator

The GEDDEMOGEN program allows users to create their own demonstration files in addition to the ones provided. During the demo creation phase two terminals are used. The first terminal runs the GEDIT program, the second is used to input the captions that will be displayed when the demo is performed.

4.3 GEDDEMO - Demo Performer

The GEDDEMO program performs a demo. It uses files that are generated by GEDDEMOGEN. During a demo performance, the user can observe the operations being performed and can control the pace of the demo play back. Typically each key stroke is explained before it is performed. This provides the user with an immediate example of how to use the functions to perform common or complex operations.

4.4 GEDBOXES - Flowchart Box and Font Generator

The GEDBOXES program allows additional flowchart boxes to be generated and merged with the text fonts.

4.5 GEDPLOT - Plot Program

The GEDPLOT program plots the drawing files. Usually it is scheduled directly from GEDIT, but it may be used in a stand-alone fashion.

5.0 Development Philosophy

In the course of any software project or as part of any software environment, needs for specialized tools are continually generated. Usually the most useful tools are simple in concept and consume a modest amount of development effort. It makes sense to develop these tools as they are needed rather than suffering without them. However, it is often the case that tool development is neglected until after the suffering and resulting frustration have taken their toll on the patience of the programmer.

Tool building was an important motivational factor behind the development of GEDIT. GEDIT itself evolved from a program named FLOW, a tool developed to generate simple flow charts.

5.1 Development Tools

Several tools were identified and developed specifically to support GEDIT. The following sections briefly describe the tools and how they are used.

5.1.1 SERCH program

The first tool developed was the SERCH program. At the time, GEDIT consisted of several hundred C source files and 20 odd different data structure include files. Whenever it was necessary to modify a data structure, so began the laborious task of identifying each source file that utilized the data structure so that it could be modified and recompiled. The result of neglecting any source file resulted in errors that were very difficult and time consuming to find.

The solution was to develop a program that given a command file containing all the files to search, would generate a listing of string occurrences cross referenced with the file names containing them.

5.1.2 MAKEREPORT Program

The MAKEREPORT program was developed to maintain current program source listings and generate command files. The core of the MAKEREPORT program is the data base file. Each line in the database file contains the name of each file, a time stamp, and attribute fields. When the update option is specified, the MAKEREPORT program scans each source file for an edit time stamp, e.g. <860612.2040>, and copies it to the data base file. If a report option is specified, the MAKEREPORT program lists each file in the data base file that contains an attribute that matches one of the attributes specified in its run string.

5.1.2.1 Maintaining Current Listings

The source files for GEDIT have several listing attributes. To generate the initial master listing, MAKEREPORT is executed with the update option and with the report attribute that specifies all source files. All the time stamps in the data base file are updated and each source file name is included in the report file. The report file is then used by a listing program to print out all the source files.

As development continues, any source files that are edited have their time stamps changed by the EDIT program. New subroutines that are written also have entries added to the data base file. When a current listing is desired, only new files or files whose contents have been modified should be printed out. To generate a report file to print out just these files requires MAKEREPORT to be run with the update and time stamp mismatch options and with the attribute that specifies all source files. Files whose time stamp does not match the data base file's time stamp have their time stamps updated and have their name included in the report file. The report file is then used by a listing program to print out the new and modified files.

5.1.2.2 Generating Command Files

Another use of MAKEREPORT is to generate a MERGE program command file for constructing relocatable library files. In the data base file, each relocatable file would contain one or more attributes that specify in which libraries they should be contained. Executing MAKEREPORT and specifying the desired library attribute would generate a report file containing the name of each file with that attribute. Next MERGE would be run using the report file as its command file. As new routines are written during the course of development, their names are added to the data base file.

5.1.3 LIST5 Program

The LIST5 program was developed for library maintenance. The LIST5 program is an interactive type 5 file, i.e. relocatable file, report program.

LIST5 can list the NAM records of each relocatable module. This allows quick checks to be made of which version of a subroutine is actually in a library. Modules from an external library can be extracted for inclusion into a local library. If a subroutine needs to have its calling sequence changed, e.g. the number of parameters increases, LIST5 can generate a listing of all modules which call the subroutine. The subroutines can then be modified and recompiled.

5.1.4 CRRE5 Program

The CRRE5 program generates a cross reference listing of type 5 file entry points and external references. For each symbol, the modules defining the symbol and the names of the relocatable files containing the modules which define the symbol are listed. In addition, indented lists of modules referencing and reference by the symbol's defining modules are created. The resulting invocation trees can be used to identify which subroutines call a module in which the calling sequence has been modified, so that the modules can be updated and recompiled.

5.1.5 CRREF Program

The CRREF program is similar to the CRRE5 program and is used in many of the same ways. It scans C source files instead of relocatable files. Each routine has the source files which define the routine listed. In addition, indented lists of the routines which the routine references or is referenced by are created.

6.0 Program Implementation - Segmentation and EMA

The non-CDS version of the HP-1000 limits the partition size to 32 pages. Thus it was necessary to rely on segmentation and EMA to support the large amount of code and data needed.

6.1 Program Composition

GEDIT is composed of a main program and 29 segments, and is written almost entirely in C, a high level structured language. The residue is written in assembly. GEDIT is composed of 497 C source files, 52 C typedef include files, and 34 macro files.

6.2 Modularized I/O

Modularizing the input and output routines resulted in many advantages. A "DEVICE" structure is used to control the output to the graphics device and to keep track of the terminal state. This allows redundant calls that change pen numbers, line types, and graphic modes to be eliminated.

6.2.1 Output

The graphics plotting sequence can also be optimized. If it is known that the terminal is currently processing a graphics plotting sequence, it is unnecessary to retransmit the plotting sequence preamble for each vector. This can result in a factor of 2 improvement in vector drawing speed. Keeping track of the current pen position allows the binary short incremental format can be used whenever

possible. This can result in up to another factor of 2 improvement in vector drawing speed.

6.2.2 Input

The `userinput()` subroutine is the main interface between GEDIT and the user. It is responsible for determining which menu key is pressed, reading the form contents and performing the appropriate class I/O during demo generation and demo performance.

6.2.2.1 Demo Generations and Performances

During demo generations, each time the `userinput()` routine is called, it first sends to GEDDEMOGEN, via class I/O, which softkey was pressed. Next, `userinput()` sends the form contents, and the cursor position read from the terminal. GEDDEMOGEN copies this information to the demo file and prompts the user at a second terminal for caption data, which is also written to the demo file.

During demo performances, each time the `userinput()` routine is called, GEDDEMO first reads the caption information from the demo file and displays it on the terminal screen and then waits for the user to press the return key. Next GEDDEMO sends the softkey number, form contents, and cursor position, via class I/O, to the `userinput()` routine. The `userinput()` routine displays the form data in the form and otherwise behaves as if the input data originated from the terminal and not from GEDDEMO.

6.2.2.2 Edit Logging and Replaying Edits

The demo capability also allows edit logging as an option. When in edit logging mode, the GEDIT and GEDDEMOGEN programs behave as if a demo generation is being performed except that caption information is not captured from a second terminal. This allows all the commands initiated by a user to be saved in a file. If some catastrophe should halt the computer, or abort the GEDIT program, the saved commands could be replayed at a later time, allowing the drawing to be regenerated. A user may also decide, at the end of an edit session, to replay the edits up to an intermediate point, thus recovering the drawing at some intermediate stage.

7.0 Summary

GEDIT suits the needs of both the novice and experienced graphic user. The menu driven interface, on-line help, and demo performance capability all contribute to making GEDIT easy to learn and use. The rich set of entities, attributes and powerful operations provide even the experienced user with a highly efficient and sophisticated tool.

QUALITY ASSESSMENT OF HP RTE SYSTEMS

Chris Smith, Bruce Campbell, Craig Fuget
11000 Wolfe Rd.
Cupertino, CA 95014

ABSTRACT

As a result of Data Systems Division's emphasis on continued customer satisfaction, the Systems Certification function of DSD's Software Quality Engineering Department has been recently expanded. The increased attention given to quality issues related to mature code has led to the development of a new testing model for the DSD's RTE PCO's. The model includes:

- Testing functions and goals,
- Department and divisional responsibilities and dependencies, and
- Quality goal assessment methods.

As an important side issue, there has been a new emphasis on the systems approach to DSD software. This has given rise to a cooperative approach to the certification of integrated systems software. This extensive effort and the refinement of our testing model have provided a fertile bed for learning about quality assessment. Many of these lessons are generic in nature. The emphasis of this paper will be to communicate our testing model and examples related to the refinement of the model. This information should be useful to OEM's and other systems developers.

HISTORY

Historically, the DSD Software Quality Engineering Department was primarily involved in quality assessment related to new operating system releases. This included testing at both the system¹ and functional² levels.

-
- 1 System Testing is the attempt to demonstrate how the product does not meet its objectives.
 - 2 Functional Testing is the process of attempting to find discrepancies between the program and its external specification.

Responsibility for the testing of software updates (or Product Change Orders, "PCO's") was distributed among the software laboratories. To integrate the development and test efforts of the software update cycle, a temporary PCO Program Manager was assigned for the C.83 release. SQE's involvement was limited to the testing of the New File System software on RTE-6 at the C.83 PCO. By the A.85 release, a full time Program Manager was in place. Additionally, the Software Quality Engineering Department began allocating resources to assist in the testing of PCO's. For the A.85 PCO, QE was responsible for the system level testing of both RTE-A and RTE-6.

A model was developed to use as a framework for the system test effort. This model broke the system testing into two distinct levels: 1) Stress and 2) Usability (a description of the System Testing Model will follow). For the A.85 PCO, System Testing was limited to Level 1/Stress with implementation of Level 2/Usability occurring at the DSD 4.0 PCO.

The history of RTE development and enhancement is not uncommon to the computer industry. While code development in the early years of computer manufacturing was considered a dark art, it has been evolving to the point of an engineering discipline. Some speculate that at the point at which solid data about the development process can be uniformly collected across the industry it will have become a science. The RTE systems represent a collage of software, evolved over a long period of time. This is a prevailing theme among established software producers. At some point, however, the total sum of software is much greater than its parts. That is, the developers no longer can change the code with a high degree of confidence that all affected parts are well understood. The need for a rigorous testing procedure on mature code is a common one.

TESTING MODEL

The model calls for testing in several different categories (see Figure 1). Level 1/Stress Testing focuses on the operating system and its major components. It is a characterization of the reliability of the core system. Historically, completion of this phase is signaled by an Acceptance Test.3

Level 1/Stress	Level 2/Usability
Busy System Destructive Prelim. Installation Configuration	Busy System Customer Simulation Full Installation Subsystem Compatibility Configuration

Figure 1. DSD System Testing Model

Busy System

Busy System Testing is the most familiar form of System Testing. It subjects various parts of the O.S. to large amounts of input or causes the O.S. to service a large number of processes. The goal is to verify that the interaction between O.S. modules and the integrity of the O.S. data structures remain intact under heavy system loads.

Some examples of the focus areas are:

- ⊕ Memory Management
- ⊕ Swapping
- ⊕ Process Management
- ⊕ Resource Management
- ⊕ State Processing
- ⊕ System Calls

This type of testing is accomplished through the use of test suites specifically designed and created for this purpose and applicable subsets of tests created for Functional Testing.

Destructive

Destructive Tests force the O.S. to exceed its capabilities and are a characterization of robustness as opposed to an actual test in the classical sense.

-
- 3 This Acceptance Test actually entails six CPUs (three RTE-6 systems and three RTE-A systems) each running with tests for 96 hours. The acceptance criteria is that no serious or critical defects are discovered and that no unresolved defects remain.

Although there is usually no correct or incorrect response, destructive testing does give valid insights into how the system will react in various situations. One example relates to the fact that RTE is an open architecture machine and although it may not be supported, some customers may attempt to manipulate the O.S. for a specific application for performance reasons. Information obtained via Destructive Testing yields a better understanding of what may be expected by modifying various system tables, especially if they are modified incorrectly.

Preliminary Installation

Preliminary Installation is a by-product of doing Busy System Testing. In order to use the system, it must first be installed. The purpose is to verify that load files and command files which are supplied with the O.S. work as they were intended. Although the types of problems encountered at this stage (i.e. typo's, missing comments, ...) often seem minor, it is important they be addressed prior to Level 2/Usability Testing. Also, Preliminary Installation provides valuable input for the "Generation and Installation" chapter of the Software Update Notice.

Configuration

In conjunction with Busy System Testing, Configuration Testing verifies that the system operates correctly with varying system parameters. The make-up of any given RTE system is the result of a number of tradeoffs made at generation time. What modules are generated in, how many resource numbers are used, what peripherals are used, and how many ID segments are allowed, etc. all have an effect on the structure of the system.

Obviously it is impossible to use all possible combinations of system parameters, therefore, the idea is to select a reasonable set of system configurations which take into account "real world" usage and areas of change at a given PCO. These configurations are then used during the Busy System Test execution.

LEVEL 2/USABILITY

Level 2/Usability Testing is the second half of the system test effort and focuses on the interactions between feature products, applications and the operating system. It attempts to model real world configurations. This is the last phase of testing prior to the software going to manufacturing and ends with an Installation Checkout that

is the final verification to insure that all of the pieces of the system have come together correctly. A number of customers have been involved in the final check-out during the last two major RTE PCO's.

Installation

Preliminary Installation of subsystems is accomplished via the DSD SQE Partners Program.⁴ We rely on the Partners to provide tests which they feel adequately exercise basic functionality and installation of their product.

The Installation Tests end with an Installation Check-Out which verifies that a defined set of subsystems can be installed on a base O.S. (typically a Primary System). The check-out focuses on verifying that the installation procedures are correct and adequately reflect the state of the software and media.

Customer Simulation

Customer Simulation is an attempt to model "real world" systems. The purpose is to find those problems which have not been caught by other forms of testing and are most likely to be encountered by the end user. This is a "regular use" form of testing and is accomplished by using demos and existing applications to verify correct operation of the system, and to characterize the impact to the customer of changes in the system.

Subsystem Compatibility

At DSD, our definition of compatibility continues to be directed by our customer's needs. We can however, gain some insight from reviewing the definitions and opinions of others. The IEEE 749 Standard defines compatibility to be the ability of two or more systems to exchange information.

John R. Grogan of International Computers Limited, England makes some worthwhile comments in "A Manufacturer's View".⁵ He suggests that compatibility be viewed as a

⁴ RTE subsystems such as Image and BASIC provide a subset of their tests to the System Certification Group for inclusion in the Level 2/Usability Testing which makes them DSD Partners.

many faceted concept. The components of compatibility should be ordered by importance and then mapped onto a product which a computer manufacturer can afford to offer customers and which will offer an economical computing service that will satisfy most if not all of a customer's computing needs.

He also suggests that differences between customers will make certain compatibility issues more important than others. Most of the differences between customers will fall into the following categories:

- ⊕ Configuration
- ⊕ File Structure
- ⊕ User Names
- ⊕ Accounting
- ⊕ Operation
- ⊕ Libraries
- ⊕ Support Routines
- ⊕ Recovery
- ⊕ Restart
- ⊕ Version
- ⊕ Maintenance Level
- ⊕ Work Priorities

Currently, DSD is using the IEEE 749 Standard definition as a guide to the development of Compatibility Tests. The actual focus areas were determined by a Subsystem Compatibility Survey conducted by SQE. Information concerning current compatibility issues was obtained from DSD \Online Support, Field SE's, engineers from Advanced Manufacturing Systems Operation, and engineers from subsystem groups (e.g. BASIC, Image, ...). For the purpose of System Testing at DSD, the scope of compatibility has been defined to be:

- ⊕ Size Constraints
- ⊕ Installation procedures
- ⊕ System Resource Contention (e.g. SAM, class numbers, ...)
- ⊕ Subsystem Interaction
- ⊕ Data Corruption

5 Brown, P.J. Ed., Software Portability, Cambridge Univ. Press, N.Y., 1977.

A Compatibility Matrix was developed from the results of the Subsystem Compatibility Survey. Subsystem Compatibility Testing is addressed through the use of software which exercises the interactions defined on the Compatibility Matrix. The software used to address the matrix consists of demos, customer applications and subsystem specific test packages.

Configuration

Taken from Software Distribution Center's Dec., 1984 Survey: "System configurations are so varied and complex that it is virtually impossible to produce a complete sampling for Field Review Tests. Generally speaking, inputs from the Field stated that full blown systems should be used with various data communications products. These systems should test subsystems concurrently with heavy disc I/O. Installations and generations should be accomplished with the aid of the Software Update Notices and Generation and Installation manuals to verify their validity."

Usability Configuration Testing is addressed through the use of a Customer Answer Files Test Suite. Focus areas have been identified, based on surveys by Product Marketing and SQE, and customer answer files have been obtained from the Field to address these areas. The answer files are run through the generator to verify the system can still be built or to quantify the magnitude of change in the update. These answer files provide a net in which to catch problems related to ease of upgrading (with respect to configuration) an RTE system.

USE OF TESTING MODEL

The main benefit of this model is the framework it provided for uniform data collection and analysis. In order to improve any process, you must get at least a general understanding of how it works, then measure the quality and efficiency of its sub-processes. The sub-process with the most problems is the one to attack.

For each problem logged, we try to understand which phase of the process caused the problem. If a problem is discovered in a phase other than the one which caused it, we attempt to find why it wasn't discovered earlier. We also work with R&D to determine how to prevent the problem from reoccurring.

The quality data may be utilized to improve software maintenance productivity through the incorporation of the following steps:

1. Keep track of the following major items:

A. Defects:

- ⊕ What went wrong
- ⊕ When it went wrong
- ⊕ Why it went wrong
- ⊕ What portion(s) of the software were affected
- ⊕ What was done about it (Was it really a problem or just a misunderstanding?)
- ⊕ Who found and who resolved the problem
- ⊕ How long it took to address the problem

B. Software Revisions:

- ⊕ How many occurred during the test phases
- ⊕ How many defects each revision had
- ⊕ When the revisions started

C. Time investment: how long the testing took in months and engineering time.

2. Correlate the data to determine what the biggest problem areas were (We've found Pareto Charts to be quite useful.).

3. Ask why problems occurred. Was it:

- ⊕ Poor Training
- ⊕ Poor Documentation
- ⊕ Poor Communications
- ⊕ Lack of Source Control
- ⊕ etc.

4. Prepare a plan to attack the causes of the problems, not just the problems themselves.

5. Ensure that management in all affected areas are aware of the problems and the potential solutions. (We present a full report at the end of each project to the project management.)

SUMMARY

We at DSD have found that by carefully evaluating the data we already had, we could gain additional insights into a time and resource intensive process without having to affect the process itself. This data allows us to learn from the problems at each phase, and to address new quality issues rather than old ones. We are excited about the challenge of exceeding customer quality expectations. and are committed to servicing our existing RTE customer base with products which match H-P's high standard of excellence.

DESIGNING AND IMPLEMENTING A COMMON SYSTEM FOR THE DEVELOPMENT OF LARGE APPLICATION PACKAGES

Stephen C. Fullerton
Statware, Inc.
P.O. Box 510881
Salt Lake City, UT 84151

INTRODUCTION

The development of large application packages usually requires many man-years for the entire cycle of design, development, implementation, and testing to be completed. This process is usually repeated for the development of each individual package. Our goal was to design and implement a development system that would permit a much easier and rapid generation of application packages.

Our system is not an application generator and should not be compared to a 4GL system. Rather it provides us with the basis for an entire family of application packages. This basis includes: memory management, common I/O, exchange of data between applications, common user interface, etc. When creating a new application package, we need only design each command and write its specific code. No parsing code need be written, the commands are described in an English-like grammar that is processed by the system and causes each command to be completely parsed before it is ever executed.

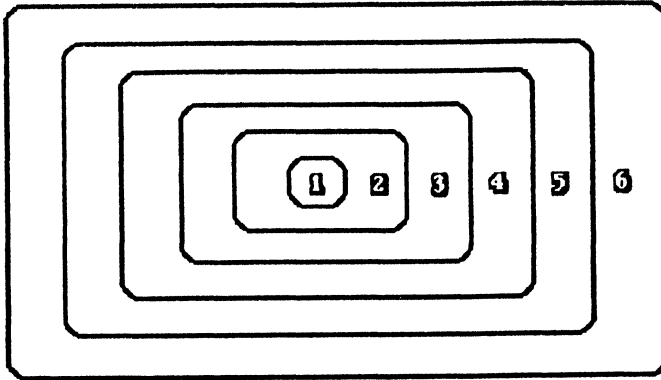
STAT80 [1], Release 3.0, was developed using this system and since STAT80 is a dynamic package, subsequent releases will be much more frequent and much easier to port to other machines.

This paper will focus on the different modules of this development system and demonstrate how an application can be designed and written.

FRAMEWORK

Our development system [2] is a multi-tiered system that is partially modeled after the UNIX [2] system libraries and system calls. Level 1 is the host operating system. Level 2 contains the system calls to the host operating systems for various system dependent operations; e.g., file handling, date and time processing, etc. This includes all low level I/O operations. Level 3 contains memory management and the library I/O operations. Level 4 contains the high level I/O operations and the command history stack. Level 5 contains the processing code for the grammar and the various parsing modules. Finally, Level 6 contains the command shell for the package with a proc facility built-in.

The following diagram illustrates this layering effect. All subroutine and function calls are made inward so at each level the number of core routines increases.



DEVELOPMENT LANGUAGE

The production version of our development system is written entirely in Ratfor, a RAtional FORtran preprocessor [3]. Ratfor is a very common preprocessor; however, it does suffer from quite a variety of dialects. We are using the Ratfor in use on UNIX and GCOS systems. This Ratfor provides:

- statement grouping
- if-else and switch for decision making
- while, for, do, and repeat-until for looping
- break and next for controlling loop exits
- free form input (multiple statements/line, automatic continuation)
- translation of >, >=, etc., into .GT., .GE., etc
- return(expression) statement for functions
- define statement for symbolic parameters
- include statement for including source files

Most dialects of Ratfor provide these features and often many more. However, the UNIX version of Ratfor [4] also supports the logical operators used in the C programming language [4].

The Sftran3 [5] preprocessor was also evaluated, but rejected due to its lack of similarity with the C programming language. A project is currently underway translating our entire system to C. Since it is in a dialect of Ratfor that is very similar to C, the work is progressing rapidly.

LEVEL 1: HOST OPERATING SYSTEM

Our system is designed to be independent of the host operating system. The entire interface to the host operating system is done with the Level 2 routines. As an example of the degree of operating system independence, STAT80 Release 3.0 consists of over 100,000 lines of Ratfor source code and was completely converted to the HP 1000 RTE-A operating system in less than two man-weeks. In contrast, STAT80 Release 2.9k consisted of over 90,000 lines of PFORT [6] verified FORTRAN source code and required two months to be converted to the HP 1000 RTE-A system [7].

LEVEL 2: OPERATING SYSTEM INTERFACE

Level 2 of our system is closely modeled after the UNIX level 2 system library. This collection of routines are the only system dependent routines in our system and must be customized for each operating system. This level consists of I/O routines, date and time routines, and character handling routines.

I/O Routines

The following table lists the names and a brief description of the system dependent I/O routines.

Level 2 I/O Routines

<u>Name</u>	<u>Description</u>
syscls	close a file
sysdel	delete a file
sysopn	open a file
sysrbb	read byte buffer in binary
sysrbd	read double precision buffer in binary
sysrbi	read integer buffer in binary
sysrbr	read real buffer in binary
sysred	read byte buffer
syssek	seek in a file
syswbb	write byte buffer in binary
syswbd	write double precision buffer in binary
syswbi	write integer buffer in binary
syswbr	write real buffer in binary
syswrt	write byte buffer

The implementation of these routines varies greatly from system to system. For example, on the HP 1000 these routines use FMP, XREIO, and EXEC. On the HP 3000, the system intrinsics are used. And on UNIX systems, the level 2 kernel routines are used.

Every attempt is made to avoid using the FORTRAN I/O statements to implement these routines. However, if necessary, a version of these routines is available that uses FORTRAN I/O as a basis. The only real loss is in the overall performance of the application.

Miscellaneous Routines

The following table lists the names and a brief description of various other system dependent routines.

Level 2 Miscellaneous Routines

<u>Name</u>	<u>Description</u>
date	return current date
time	return current time
abts80	abort application immediately with message
ilmach	machine dependent integer parameters
rlmach	machine dependent real parameters

The date and time is usually easy to determine regardless of the operating system. The routines `ilmach` and `rlmach` are part of the PORT Library Framework [8] and supply the machine dependent parameters for each operating system; e.g., number of bits per integer unit, number base of the machine, largest relative spacing, etc.

Character Primitives

The following table lists the names and a brief description of the system dependent character handling routines.

Level 2 Character Primitives

<u>Name</u>	<u>Description</u>
chrasu	upper case ASCII ordinate of character*1 character
karasc	ASCII ordinate of character
karasu	upper case ASCII ordinate of character
karchr	character of ASCII ordinate
karcn2	string comparison ignoring case
karcnp	string comparison
karget	get character from packed string
karid2	substring index ignoring case
karidx	substring index
karlc	lower case character
karlcl	local character of ASCII ordinate
karmov	move characters from packed strings

Level 2 Character Primitives

<u>Name</u>	<u>Description</u>
karord	local ordinate of character
karpak	pack characters
karpuk	put character into packed string
karuc	upper case character
karupk	unpack characters
karvfy	verify characters
karxlt	translate characters
krctoi	character to packed string
krgetc	character get from packed string
krgeto	character get returning ASCII ordinate
krgetu	character get returning upper case ASCII ordinate
kritoc	packed string to character
krputo	put ASCII ordinate into packed string
lentrn	length of character string

We use the term Hollerith rather loosely. Rather than using the FORTRAN [9] character data type which has many shortcomings, we have adopted the Pascal methodology of storing character strings. All strings are stored packed into integer arrays with each string preceded by its length as a single integer. This was necessary in order to comply with our memory management rules discussed in the next section.

No direct comparison of characters is ever made, rather all compares and lookups deal with the ordinal value of each character (ASCII ordinate).

LEVEL 3: MEMORY MANAGEMENT AND LIBRARY I/O

Memory Management

Effective memory management is one of the major features of our development system. Since FORTRAN does not support dynamic allocation of memory or a pointer data type, we designed and implemented our own portable memory management scheme.

Our memory routines manage a heap that contains a linked list of free and allocated blocks of memory. The heap is maintained in a FORTRAN common block. In order for dynamic memory allocation to be useful, it must support byte, integer, real, and double precision data. Complex data might also be useful and could be easily added to our system.

At the beginning of the memory block is the link to the next block and at the end of the block is a check digit. This digit is tested when the block is released and if corrupted, the application will abort. This is very useful during the development phase as it quickly locates memory

boundary errors.

The memory management routines handle data types, BYTE, INT, REAL, and DOUBLE. In order to avoid alignment problems with machines with different lengths for INT and REAL, all allocation is done by alignment to the boundary of the largest unit; i.e., DOUBLE. This is unfortunate in that small allocations have extra overhead; however, it is necessary for portability.

The following is the Ratfor include that contains the FORTRAN common block and other definitions for our memory management routines. All routines using the memory management must include these statements.

```
#
#      @(#) memory.h 1.5 10/15/85 16:56:42
#
#      Memory common block (used by memory handling routines)
#
define(MEMINT,30000)
define(MEMREAL,30000)
define(MEMDBLE,15000)
define(MEMSIZE,15000)
#
define(BYTE,1)
define(INT,2)
define(REAL,3)
define(DOUBLE,4)
#
      integer      maxmem, imemry(MEMINT),  nxsrch
      real         rmemry(MEMREAL)
      double precision dmemry(MEMDBLE)
#
      common / s80cb6 / maxmem,      nxsrch, dmemry
      equivalence  (dmemry(1),rmemry(1),imemry(1))
#
```

Note the absence of the FORTRAN character data type. This is because of two major problems: first, character variables cannot be equivalenced to non-character items. And second, the ANSI standard [9] fails to define a minimum or maximum limit of the length of a character variable. Some FORTRAN compilers do not place restrictions on either of these cases; however, many others choose an arbitrary limit. Character strings in our system are stored packed into integers similar to the string storage in the Pascal programming language.

The following table lists the names and a brief description of the

memory management routines.

Level 3 Memory Management

<u>Name</u>	<u>Description</u>
malloc	allocate a block of memory
mallom	allocate the largest block of memory
mfree	release a block of memory
mpartf	release partial block of memory
mgarbc	garbage collection
memmap	histogram of memory usage
memerr	display memory allocation error message
minit	memory management initialization

We used a naming convention very similar to the UNIX memory management routines used in the C programming language. Function malloc returns an index to the requested block of memory. For example,

```
np = malloc(100,INT);
```

sets the variable, np, to be used as an index into imemry() for the requested 100 words of memory. That is,

```
imemry(np)      = word 1
imemry(np+1)    = word 2
imemry(np+2)    = word 3
.               .
.               .
imemry(np+98)   = word 99
imemry(np+99)   = word 100
```

Analogously, real (floating-point) memory is allocated as:

```
nr = malloc(50,REAL);
```

This sets the variable, nr, to be used as an index into rmemry() as follows:

```
rmemry(nr)      = word 1
rmemry(nr+1)    = word 2
.               .
.               .
rmemry(nr+48)   = word 49
rmemry(nr+49)   = word 50
```

Note that this scheme is essentially a pointer to a base-zero array, just as it is done in the C programming language.

The byte data type is different in that the malloc routine returns an index into imemry() with enough memory to contain the packed byte stream.

The memory is released by calling the mfree subroutine as follows:

```
call mfree(np,INT);  
call mfree(nr,REAL);
```

The allocation is done by searching the free list for either an exact match or for the smallest block that is large enough. This type of search avoids fragmentation of the heap. If a large enough block isn't found, then the garbage collection routine is called. It performs only a simple collection by connecting adjacent free blocks of memory to be a single large block. If there still isn't a large enough block, then the return value is negative, with its magnitude being the largest available free block.

All of the higher levels in our development system rely upon these memory management routines. Therefore, they must be simple, efficient, and reliable.

Library I/O

The library I/O routines have been copied in form and functionality from the UNIX level 3 I/O functions. However, all of our applications have the capability for the output to be sent to both the terminal and a paginated log file. For this reason, all of the printing routines support two file pointers rather than one. Also, terminal pagination with forgiving interrupt capability is also desired. At the end of a terminal screen, this prompt is displayed:

More?

Entering a RETURN will cause the application to resume and display the next screen. Other valid responses are:

Valid Responses to "More?" Prompt

<u>User Enters</u>	<u>Application Responds With</u>
RETURN	Next screen of output.
y	Next screen of output.
.	Next line of output.
c	Continuous output until finished.
n	Suppress terminal output, processing continues
Break Character	Abort current command
Abort Character	Session aborts; application exits.
?	Display possible responses

The "Y" response is identical to a RETURN, "." will only display the next line of output, and "C" will continue the output of the command without terminal pagination. The "N" response will suppress terminal output; however, processing of the command continues along with output to the log file, if any. The break character, usually an upper case "B", will cause the break flag to be set. This will interrupt output to both the terminal and the log file. The abort character, usually an upper case "A", will cause the application to abort immediately.

All of the popular I/O functions from UNIX and C [4] are supported along with many more. Analogous to the printf() family of functions in C are the functions: prtfc, prtff, prtfh, prtfi, prtfm, mprtff, mprtfi, and mprtfm. Since FORTRAN doesn't support routines with a variable number of arguments and types, each of these routines supports only one argument and only one type. For example, to print a floating-point value, rval, enter:

```
call prtff(stdout,NULLFP,'Value = %f&n',rval);
```

The output is to 1 or 2 file pointers, either to a pre-defined file pointer such as stdout and stderr, or one set by the fopen function. NULLFP is the defined parameter for a null file pointer; i.e., not set. The conversion string, 'Value = %f&n', controls how the value is output. The conversion string is similar to that of the C programming language; however, many more conversion characters are supported. Also, if the field width or precision isn't specified, then only the significant digits of the value will be output. This makes for much cleaner reporting.

The following table lists most of the available I/O routines:

Level 3 Library I/O

<u>Name</u>	<u>Description</u>
ctof	decode floating-point from character
ctoi	decode integer from character, base 2..16
decstr	decode character string
fclose	file close
ferror	report file error
fflush	flush buffer
fgets	read string
finit	file I/O initialization
fopen	file open
fputc	put character to file pointer
fputc	put character string to file pointer
fputo	put ASCII ordinate to file pointer
fputs	put packed string to file pointer
fread	read binary
frewnd	rewind file

Level 3 Library I/O

<u>Name</u>	<u>Description</u>
fseek	file position
fterm	file I/O termination
ftoc	code floating-point in character form
ftoi10	floating-point to integer array, base 10
fwrite	write binary
getc	get character
geto	get ASCII ordinate
itoc	integer to character form
itocu	integer to character form, unpacked
itor	integer to roman numerals
mprtff	print floating-point formatted, format in packed string
mprtfi	print integer formatted, format in packed string
mprtfm	print packed string formatted, format in packed string
perrc	print error message, character value
perrf	print error message, floating-point value
perrh	print error message, Hollerith value
perri	print error message, integer value
perfm	print error message, packed string
pffmt	print floating-point with F-type format
pftoc	print floating-point
pictfc	print floating-point using picture format
pictfh	Hollerith front-end to pictfc()
pictfm	packed string front-end to pictfc()
pitoc	print integer
pitor	print integer as roman numerals
prtfc	print character string formatted
prtff	print floating-point formatted
prtfh	print Hollerith string formatted
prtfi	print integer formatted
prtfm	print packed string formatted
prttcp	print to column position
prtt8	print to 8 column tab stop
putc	put character
putcs	put character string
puto	put ASCII ordinate
puts	put string
readrc	read direct access
termpg	terminal pagination
termrs	terminal reset
ungetc	unget character
ungeto	unget ASCII ordinate
writrc	write direct access

LEVEL 4: HIGH LEVEL I/O AND HISTORY STACK

Level 4 is the first level in our system that goes beyond the current capabilities of most development systems. That is, levels 1-3 provide an operating system interface, low level I/O, memory management, character handling, and a complete library of I/O routines. Much of this is provided in programming languages; e.g., C. However, in order to maintain a homogeneous development system with a high degree of portability, we had to design and implement these levels without regard to the language. Even in our C version of our system, we still implemented all of these levels without using the UNIX libraries.

All of our applications support a higher level of I/O; i.e., line continuation, in-line comments, an escape mechanism for special characters, and a history stack. This requires yet another level of I/O routines to be layered in between the application and the library I/O routines.

The additional routines provide each of these capabilities transparently to the application. For example, the history stack is available to every command that does terminal input. Our history stack is modeled after the command history in the UNIX csh (C shell).

The following are valid history commands for using the history stack, assuming that the history character is set to '!':

<u>Command</u>	<u>Meaning</u>
/*	Display command stack.
//	Retrieve last command.
/n	Retrieve command number "n".
/-i	Retrieve command "i" back on stack.
/s	Retrieve last command starting with string "s".

The retrieved command may be modified by the substitution of one string of characters for another string, or by appending additional information to the command.

Additional information is appended by entering it after the retrieval command, delimited by a single space. This may also be used in conjunction with a substitution modifier, when the additional information is entered after the substitution modifier.

Substitution of strings is indicated by one of the following modifiers, assuming that the history delimiter character is ":".

ModifierMeaning

:s	Single substitute.
:g	Global substitute.
:p	Single substitute, put back on stack.
:q	Global substitute, put back on stack.
:r	Substitute the character "?".

The first character to follow the :s, :g, or :p modifier is used as the string delimiting character. The first string, set apart with this character, will be substituted with the second string, again set apart with this character. For example,

```
/22:s'/file'/nolist'
```

means that in the retrieved command (#22), the string "/file" is to be replaced with the string "/nolist" before the command is passed to the application for execution. The apostrophe, being the first character following the substitution modifier, is used as the string delimiter. An unmatched delimiter would result in an error. Other examples:

```
//:g$v3$v2$  
/print:p/all/v2 to v4,v7/
```

The substitution modifiers differ as summarized in the above table. The ":s" modifier will substitute the first occurrence of the first string. The ":g" modifier will substitute all occurrences of the first string. The ":p" modifier will do a single substitution and put the command on the top of the history stack without execution; this is used when multiple substitutions are required and cannot be done with one command. The ":q" modifier will substitute all occurrences of the first string and put the command on the top of the history stack as does the ":p" modifier. The ":r" modifier will replace the in-line help character, "?", with whatever is following this command.

Suppose the last command on the history stack is:

```
HISTOGRAM V3 /SELECT=(V3 > 5)
```

Then the the user issues this history command:

```
//:g'v3'v4' /CUM
```

The breakdown of this command is as follows:

```
//  -- retrieve the last command from the command stack
:  -- delimiter, indicating a substitution command
g  -- global substitution
'  -- string delimiter
V3 -- first string
'  -- string delimiter
V4 -- second string
'  -- string delimiter, final
/CUM -- append this text to the command
```

Thus, the resulting command which is submitted for execution is:

```
HISTOGRAM V4 /SELECT=(V4 > 5) /CUM
```

LEVEL 5: GRAMMAR DEFINITION AND PARSING

Level 5 is the beginning of the more complex modules of our system. At this point all of the lower level code has been defined and implemented that will support a very large range of applications; however, a great deal of effort would be required in order to create a new application using only levels 1-4.

In this level we define a grammar that allows the developer to design and test command form without having to write any code. Furthermore, our grammar does a complete parse of the input commands including all arithmetic, logical, and matrix expressions. Currently, arithmetic expressions have 120 functions and 15 operators available and matrix expressions have 51 functions and 18 operators available. All of this can be incorporated into the application without developing any new code.

Our grammar is not defined in a Backus-Naur Form (BNF) and not in a form typically used by compiler-compilers such as YACC (Yet Another Compiler Compiler) [10]. Rather, we are developing applications that use data stored either as bytes, integers, reals, or double precision in such entities as scalars, columns, tables, vectors, and matrices. There was no reason to begin a grammar definition at such a low level since we already know a great deal about the objects manipulated in our applications.

Another reason for NOT going the route of a parser-generator or compiler-compiler is that each of our applications are extensible; that

is, the user can easily add new commands and procedures dynamically. Our language supports looping, branching, statement grouping, etc., so that the user can create new commands using this language. In order for the new command to be integrated into our applications, we provide the user with a grammar to define the command form. This grammar is the same one we use to develop the application.

The following is a simplified example of our grammar:

```
PRINT:97 @C<PRINT V4 to V8> <V,E,RC,CD>@C<Variable(s) to print>
```

The command name is PRINT and is identified to the application by the id number 97. Following the command name is an example of the command in use: @C<PRINT V4 to V8>. This can be displayed whenever a user requests an example of the PRINT command. The next item tells the application that a variable list is required: <V,E,RC,CD>. The "V" tells the application that a variable list is needed, the "E" means that the variables must exist, the "RC" means that they may be either real (floating-point) or character, the "CD" means that either a column variable or a dummy variable (our terminology for a scalar variable) may be printed. The last item, @C<Variable(s) to print>, will be displayed when the user enters:

```
PRINT ?
```

requesting in-line help.

In a nutshell, our grammar provides us with a means for quickly designing new commands and modifying existing commands. We have also built an in-line help system into the grammar. This was quite easy since the grammar directs the application as to what is needed from the user; therefore, if a "?" is encountered, just display this information to the user. Furthermore, whenever an error occurs, the grammar contains the information to alert the user to what was expected.

The following table lists the different grammar terminal symbols (identifiers) with a brief description:

Grammar Symbols

<u>Symbol</u>	<u>Example</u>	<u>Definition</u>
NAME or 'name'	Cases or '!='	Literal, must be specified by the user
/NAME	/Classes	Option name
<V>	<V,E,RC>	Variable list, 1 or more variables
<V#>	<V1>	Fixed variable list, 1 to # variables
<R>	<R>	Real value list, 1 or more values
<R#>	<R3>	Fixed real value list, 1 to # values
<I>	<I>	Integer value list, 1 or more values
<I#>	<I5>	Fixed integer value list, 1 to # values

Grammar Symbols

<u>Symbol</u>	<u>Example</u>	<u>Definition</u>
<S>	<S>	String list, 1 or more strings
<S#>	<S2>	Fixed string list, 1 to # strings
<F>	<F>	File name, blank delimited name
<A>	<A>	Anything, 0 or more words (items)
<\$>	<\$>	Integer temporary variable
<#>	<#>	Real temporary variable
<%>	<%>	String (character) temporary variable
<T>	<T>	Token list, 1 or more tokens (words or items)
<T#>	<T1>	Token list, 1 to # tokens (words or items)
<E>	<E>	Arithmetic or logical expression
<M>	<M>	Matrix expression
@C<...>	@C<Dependent>	Comment for in-line help
@O<	@O<	Begin optional list
@O>	@O>	End optional list
@I<	@I<	Begin index list
@I>	@I>	End index list

LEVEL 6: PACKAGE SHELL AND PROC FACILITY

All that is needed at this time is a driver program for all of the lower level routines. The grammar definition and parsing code enables the user to define from 1 to 196 commands for the application. Each command is identified by a numerical command id which is in the range from 1-196. Recall the example command described in the previous section:

```
PRINT:97 @C<PRINT V4 to V8> <V,E,RC,CD>@C<Variable(s) to print>
```

The command id for this command is 97. Our package shell has calls for subroutines CMD001-CMD196 already installed. When the user correctly enters the PRINT command, the package shell calls CMD097. Common blocks are provided via Ratfor include statements to provide the command routine with all of the parsing information; e.g., variable names, options, etc.

Therefore, all that is really needed in order to produce a new application is to describe the commands in the grammar and then write the routines for the individual commands. Everything else is already provided.

Our applications also support what we refer to as a "proc" facility. A "proc" is a command or procedure that is written by the user and is executed under the control of the application. The language is called

the "proc" language and contains control statements such as IF-THEN/ELSE, WHILE-ENDWHILE, REPEAT-UNTIL, FOR-ENDFOR, DO-ENDDO, DO CASE-CASE-ENDCASE, GOTO, GOSUB, etc. The command form for these procs is written using the same grammar as all of the built-in commands. An internal compiler is also available that processes these procs so that they become integrated with the application.

References

1. STAT80 User's Guide, Statware, Inc., P.O. Box 510881, Salt Lake City, UT 84151 (1986).
2. UNIX Programmer's Manual, Bell Telephone Laboratories, Inc. Murray Hill, New Jersey, Holt, Rinehart and Winston.
3. Kernighan, Brian W., RATFOR - A Preprocessor for a Rational Fortran, Software - Practice and Experience 5, 395 (1975).
4. Kernighan, Brian W., Ritchie, Dennis M., The C Programming Language, Bell Laboratories, Murray Hill, New Jersey, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.
5. Lawson, C.L. and Flynn, J.A., Sftran3 Programmer's Reference Manual, JPL Document No. 1846-98, December 1, 1978, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California.
6. Ryder, B.G. and Hall, A.D., The PFORT Verifier, Software Practice and Experience 4, No. 4, October-December 1974, pp. 359-377.
7. RTE-A Programmer's Reference Manual, Hewlett-Packard, Data Systems Division, 11000 Wolfe Road, Cupertino, CA 95014, Part No. 92077-90007.
8. Fox, P.A., Hall, A.D., and Schryer, N.L., Algorithm 528. Framework for a Portable Library, ACM Transactions on Mathematical Software 4, 177-188 (1978).
9. American National Standards Committee, American National Standard Programming Language FORTRAN (FORTRAN 77), Document X3.9-1978, American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018.
10. Johnson, S.C., YACC (Yet Another Compiler Compiler), Computing Science Technical Report #32, July 1975.

Effective Use Of Tools And Programming Style In Managing Major Software Systems

Mathieu Federspiel
Statware
P.O. Box 510881
Salt Lake City, UT 84151

Introduction

Major software systems pose special programming problems for software companies. These problems include portability, documentation, ease of modification, documentation of changes, and training of new programmers. Many aspects of what is considered "good programming practice" help to reduce the impact of these problems. These practices are good, but the needs of our company exceed the problem solution capability provided by these practices. We have implemented procedures which require the development of structured programs, utilize modular code with multiple levels, enforce extensive programmer documentation of code, and utilize tools to record changes in the code and produce formatted documentation for each module.

Many articles have appeared in the literature to discuss aspects of this problem from various perspectives. In general, portability and ease of modification of major software systems is achieved by enforcement of programming standards. These standards may include modular programming, structured coding, and documentation standards.

At Statware, we have implemented coding and documentation standards which are followed by our programmers. We have written software tools to aid in our documentation, and use software administrative tools provided as part of our Unix operating environment. The use of software tools not only aids our documentation, but enforces programming standards by requiring specific information in the source code in a restricted format.

This paper focuses on one of our in-house software tools, ABST. As the use of this tool requires the use of other tools and the use of coding standards, the discussion will also enter these areas.

Problem and Solution

Statware develops the statistical software product STAT80™. A staff of 4 programmers maintain the source code and documentation. The source code consists of over 600 Ratfor modules.

The problem is how to keep all programmers current of software changes. Changes occur during bug fixes and the addition of enhancements to the code. When a change is made in a module, it must be documented to all programmers who may use that module.

The solution used at Statware is an in-house tool used to create an abstract for each module. This abstract must provide adequate documentation for a programmer to use the module correctly without looking at the source code. The abstract is used as the first reference to the module and its function. The programmer may quickly note changes in the abstract, and check all modules which reference the modified module.

The program we use is called ABST. ABST is a C program, and follows standard Unix program invocation. At invocation, the program name is followed by several options and a list of files to be processed. Output is written to standard output.

An example of ABST output follows. This output was created with the command:

```
abst -m -t "I/O ROUTINES" -r "1.2" fopen.r > fopen.ab
```

The output file, **fopen.ab**, was processed through mm(1) with the command:

```
mm -T450-12 -rW80 fopen.ab > fopen.ab.mm
```

Pipes may be used to do the complete processing with one command line.

NAME

fopen -- file open

SCCS

@(#) fopen.r 1.8 10/24/85 11:48:50

SYNOPSIS

integer function fopen (filnam,mode,recsiz,bufsiz,access,blksiz,mfsize,
statu⁸)

character*(*) filnam
integer mode
integer recsiz
integer bufsiz
integer access
integer blksiz
integer mfsize
integer status

DESCRIPTION

Routine to handle all of the file opening of STAT80 (including standard input, standard output, and standard error).

Fopen() allocates a block of memory and uses the following structure:

fpoint(fp): file number (system dependent)
fmodes(fp): mode (0-r, 1-w, 2-r/w, 3-a, etc)
fname(fp): length of file name in bytes
fnames(1,fp): file name (packed)
frepsz(fp): record length (may not be used on some systems)
fbuflsz(fp): buffer length (can be one for unbuffered I/O)
fbuflnc(fp): number of characters in the buffer
fbuflpt(fp): pointer to current location in the buffer
faccss(fp): file access DIRECT or SEQUENTIAL
fblock(fp): block size
ffsize(fp): maximum file size for access == DIRECT
fstats(fp): file status, PERM or SCRATCH
fcolps(fp): column position
fbuffr(fp): file buffer

INCLUDE FILES

```
include files.h
include memory.h
```

EXTERNAL REFERENCES

external refs	max0,	malloc,	gtfnam,	krctoi
external refs	len,	mfree,	sysopn	

EXTERNAL FUNCTIONS

integer	max0,	malloc,	gtfnam,	len
integer	sysopn			

Output and Usage

The output produced by ABST follows the format of standard Unix manuals. The output may be an English text file, or a text file containing embedded mm(1) formatting commands. In the latter case, processing by mm(1) produces the abstract in the standard format.

The information used by ABST in creating its output is taken from the Ratfor source code. The source code for the example abstract is given in Appendix 1. Some information may be added at the time of abstract creation by the user, e.g., adding a title in the title bar with the -t option.

To enable correct parsing of the source code, standards must be adhered to. ABST requires that blocks of code containing specific information be separated by a record beginning with the character sequence "#- - - - -". Within each block, a search is made for specific types of information. Blocks are referenced by number starting with block zero. Programmers must create these blocks of code with the appropriate information for ABST to produce a correct abstract. ABST has limited warning and error capabilities for detecting source code which does not meet these standards.

Block zero of the source code contains the module declaration line and a one line comment containing a description of the function of the module. Block one contains the copyright notice and is ignored by ABST. Block two contains an extended description of the routine and the SCCS line. Blocks greater than two are checked for variable typing, external references, external functions, and include files.

ABST has seven options which permit the user to design and control output from the program. With this many options available, a built in help system is useful. ABST follows Unix standards, printing a summary usage description when invoked with no parameters. An example of this output follows with the user input is underlined:

```
% abst
Usage: abst [-m -o outfile -c -p -t title -r rev -i ifile] file-names
      -m == produce output for mm processing
           (e.g.: mm -rW80 -rO0 file > file.mm)
      -o == direct output to outfile
      -c == chatter to tty about what is happening
      -p == start each abstract on page one
      -t == put title on output
      -r == put revision in parentheses on header
      -i == dump name and summary line to ifile
      file-names == list of files to process

%
```

As an additional aid a man page is available for ABST.

Note that the options -p, -t, and -r have no effect unless the -m option is also specified. These three options use the page heading and page footer macros of the mm(1) text-formatting macro package.

Title Bar

The optional title bar at the top of the page is created with the -t option. This option places the following string as a title at the center of the top of the page. The module name, appearing in each corner of the title bar, permits easy identification of the abstract when bound with others.

The module name is obtained from the module declaration line. ABST will parse this line to extract the name from valid preceding definitions and following parameters. This line may extend over several records, as is permitted by Ratfor. No coding standards beyond those set by Ratfor need be followed by the programmer at this point.

Name

The next section of the abstract page contains the module name and a one line description. The module name, which was also used in the title bar, is obtained from the module declaration line. The one line description is obtained from the first comment line after the module declaration line. ABST requires this text be enclosed by parentheses. If not present, the abstract will not have a description following the name and a warning will be issued.

SCCS

The SCCS section contains information regarding the module's position in the SCCS system^[1]. The SCCS system is used by Statware to manage the development and changes made to the source code. This system provides the following features:

- Tracking of all changes to a module file as well as who made them and comments about what the change was.
- The current version of the source may be retrieved at any time.
- Previous versions of the source code may be retrieved at any time.
- Multiple copies of source code at different periods of development need not be kept on disk at one time.
- An SCCS System Administrator may be designated to control who may change the source code.

The information used by ABST is either the what(1) line produced by get(1), or the SCCS line entered by the programmer prior to the module being placed in the SCCS system for the first time. This line must appear in block two of the source code. If no line is found, the SCCS section on the abstract is blank.

Synopsis

The synopsis section contains the full module declaration line, followed by the variable type of each parameter passed to the module. ABST stores the name of each parameter passed to the routine and types it when parsing the variable declaration section of the source code. This important information requires no special coding by the programmer as ABST follows the rules of Ratfor.

Description

The description section contains an informative description of the action of the module. This text is taken from block two of the source code. The text in this block is written by the author of the module, and will describe what the module does, what parameters are required, and what parameters are returned. The author should make this text clear and completely descriptive to others who may need this information.

Include Files

The include files are read directly from the Ratfor source code. They are identified by the records which begin with the string "include" starting in column one. Only blocks of code after block two are checked for include files. If not include files are defined, this section is blank.

External References and External Functions

External references are expected to occur in a block of code with no other types of records. The first record which is not a blank comment record will contain the string "external references". Following will be the comment records which contain the string "external refs" followed by function, subroutine, or common variable names.

External functions are expected in a block of code similar to that for the external references. The first record which is not a blank comment record will contain the string "external functions". Following records will contain function and subroutine type definitions.

This section of code is generated by the Extended PFORT Verifier. This Verifier was originally developed by Ryder and Hall as the PFORT Verifier^[2]. The verifier is a program which parses a FORTRAN source code file to identify portability problems. Additional work was done by Beebe^[3] to make the program more tolerant of the input FORTRAN code, and this tool, the Extended PFORT Verifier, is used at Statware. The Verifier identifies machine dependencies in our code as well and produces an output file containing the external references and functions in each module, and non-common variable definitions. This file is incorporated into the source code, replacing the declarations used during the design of the module.

Either of these sections may be blank if the module has no external references or external functions.

Other Information

Variables which are local to the module are not reported in the abstract. The abstract is intended for the programmer who will be calling the module from other modules and will not be concerned with the internal workings of the module.

Sections on the abstract which are blank indicate that nothing for that section is present in the module. The section heading is kept in the abstract to indicate that the programmer need not be concerned with items in this section. The author of the module should check the abstract for completeness and as a self check that coding standards have been followed.

Summary

The management of major software systems is possible through the use of tools and programming standards. Software tools, both those written in-house and those in general distribution, may be effectively used to aid the management and development of software systems, and to enforce the programming standards which have been set. ABST has been an important tool in enforcing the standards we set for ourselves, and in providing timely information to prevent wasted development time.

References

1. SCCS, HP-UX Concepts and Tutorials Vol. 3: Programming Environment, Hewlett-Packard Company, Fort Collins, CO, 1986.
2. Ryder, B.G. and A.D. Hall, "The PFORT Verifier", Computing Science Technical Report #12, Bell Laboratories, Murray Hill, New Jersey, May 1973, revised January 1981.
3. Beebe, Nelson H.F., "The Extended PFORT Verifier", College of Science Computer Report, University of Utah, Salt Lake City, Utah, 1981.

Appendix 1: Ratfor Source Code for Example ABST Formatted Output

```

integer function fopen (filnam,mode,recsiz,bufsiz,access,blksiz,mfsize,
    status)
# (file open)
#-----
#
#   STAT80:  An Interactive Statistical Package
#
#   Copyright (C) 1985  Statware -- All Rights Reserved
#
#   Proprietary Software:  The contents of this routine shall
#   not be disclosed or made available, or any portion thereof
#   in any form whatsoever to any person other than the author
#   without prior written approval of the author.
#
#-----
#
#   Routine to handle all of the file opening of STAT80 (including
#   standard input, standard output, and standard error).
#
#   Fopen() allocates a block of memory and uses the following structure:
#
#   fpoint(fp):  file number (system dependent)
#   fmodes(fp):  mode (0-r, 1-w, 2-r/w, 3-a, etc)
#   flname(fp):  length of file name in bytes
#   fnames(1,fp):  file name (packed)
#   frecsz(fp):  record length (may not be used on some systems)
#   fbufsz(fp):  buffer length (can be one for unbuffered I/O)
#   fbufnc(fp):  number of characters in the buffer
#   fbufpt(fp):  pointer to current location in the buffer
#   faccss(fp):  file access DIRECT or SEQUENTIAL
#   fblock(fp):  block size
#   ffsz(fp):  maximum file size for access == DIRECT
#   fstats(fp):  file status, PERM or SCRATCH
#   fcolps(fp):  column position
#   fbuffr(fp):  file buffer
#
#   @(#) fopen.r 1.8 10/24/85 11:48:50
#
#-----
#
#   external references (function,subroutine,common)
#
#   external refs      max0,          mallob,      gtfnam,      krcloi
#   external refs      len,           mfree,       sysopn
#-----

```

```

#
#   external functions and subroutines
#
integer      max0,      malloc,      gtfnam,      len
integer      sysopn
#-----
#
#   non-common variables
#
sets         fp,         i,         ifptr,         lfn
sets         nblksiz,    nbufsz,    nfilsz
sets         nrecsz,     lennam
#
integer      blksiz,     bufsiz,     access,       fp
integer      i,          ifptr,      lfn,           mfsiz
integer      mode,       nblksz,     nbufsz
integer      nfilsz,     nrecsz,     lennam,       recsiz
integer      status
character*(*) filnam
#
include files.h
include memory.h
#
fioerr = 0;
lennam = len(filnam);
fp = 0;
for (i = 1; i <= maxbfs; i = i + 1) { # find open slot
    if (fbuffr(i) == 0) { # got one
        fp = i;
        break;
    }
}
if (fp != 0) { # not at maximum yet
    if (gtfnam(filnam,lennam) < 0) # bad file name
        return(-2);
    if (mode < READ || mode > APPUPD) # illegal mode
        return(-3);
    if (access != SEQUENTIAL && access != DIRECT && access != BINARY)
        return(-1);
    if (recsiz == (DEFAULTREC))
        nrecsz = RECSIZE;
    else
        nrecsz = max0(1,recsiz);
    if (bufsiz == (DEFAULTBUF))
        nbufsz = BUFSIZE;
    else

```

```

        nbufsz = max0(1,bufsiz);
    if (blksiz == (DEFAULTBLK))
        nblksz = BLKSIZE;
    else
        nblksz = blksiz;
    if (mfsiz == (DEFAULTSIZ))
        nfilsz = FILSIZE;
    else
        nfilsz = mfsiz;
    ifptr = mallob(nbufsz);
    if (ifptr <= 0) { # not enough memory
        call memerr('file open',nbufsz,ifptr,BYTE);
        return(-4);
    }
    lfn = sysopn(filnam,mode,nrecsz,nbufsz,access,nblksz,nfilsz,status);
    if (lfn < 0) { # open failed
        call mfree(ifptr,BYTE); # free buffer
        return(-5);
    }
}

#
# set up buffer
#

fpoint(fp) = lfn;          # file number (system dependent)
fmodes(fp) = mode;        # mode
fname(fp) = lennam;       # length of file name (in bytes)
call krcroi(fnames(1,fp),1,filnam,lennam); # put in filename
frecsz(fp) = nrecsz;      # record size
fbufsz(fp) = nbufsz;      # buffer size
fbufnc(fp) = 0;           # number of characters in buffer
fbufpt(fp) = 0;           # pointer to current location
faccss(fp) = access;      # file access
fblock(fp) = nblksz;      # block size
ffsize(fp) = nfilsz;      # file size
fstats(fp) = status;      # file open status
fcolps(fp) = 0;           # column position
fbuffr(fp) = ifptr;       # pointer to buffer
return(fp);
} else # max files open
    return(-6);
return;
end

```


ROBOTICS AND DATA SYSTEMS
IN THE
CHEMICAL ANALYSIS LABORATORY

Chris Scanlon
Hewlett-Packard
4326 Cascade Road, S.E.
Grand Rapids, Michigan 49506

INTRODUCTION

The tasks in the analytical chemical laboratory can be divided into three general areas:

- 1) sample preparation
- 2) sample analysis
- 3) data handling

Over the past 10 - 15 years, laboratory automation efforts have been focused primarily on sample analysis and data handling. Hewlett-Packard products for sample analysis include gas and liquid chromatographs, mass spectrometers and spectrophotometers. Data handling products include standalone integrators, single user workstations (HP 9000) and multi-user lab data systems (HP 1000). HP's 1000-based systems for data acquisition and lab data management are installed in well over 1000 laboratories world wide. More recently, the problem of automating sample preparation and handling has begun to be addressed through the use of robotics and dedicated sample handling systems such as bar code readers, autodiluters, pipetters, etc. These latest developments have provided the last pieces necessary for a totally automated laboratory bench. The challenge ahead will be to tie the automated sample preparation to the chemical instrumentation and data handling networks so that they all work in unison and can be configured to each user's particular needs. Thus, with little or no human intervention, samples can be automatically logged in for tracking, prepared and analyzed, then results and tracking information can be stored in a database. From there, final reports can be generated and passed on to printing devices and/or other more general purpose mini-computers or mainframes in the corporate information network.

The purpose of this paper is to describe how robotic sample preparation has already been integrated with Hewlett-Packard sample analysis instruments and computer systems. In two examples, a Zymark automatic sample preparation system is used to pass prepared samples to an HP gas chromatography system. In the first case, an HP 150 is configured to co-ordinate the data transfer between the sample preparation system and the analytical system and then stores the results of the analysis for later use. The second example employs an HP 1000 system that can not only co-ordinate data transfer but can also select the analysis to be performed based on the sample identification information passed to it from the robotics system, initiate the analysis, acquire raw digitized data from the analysis system, reduce the data into peak information plus chemical component identification and amounts, and produce customized reports. By adding LABSAM/1000 software either on the same system or on another system connected via DS/1000, LIMS sample management capabilities are easily integrated as well.

EQUIPMENT OVERVIEW

Both examples that will be described utilize a Zymark robotics system for automatic sample preparation and an HP 5890 gas chromatographic system for chemical analysis. Specific components are shown in figures 1 and 2.

The Zymark system includes a Z120 Zymate Controller, Z110 Robot Module, Z830 Power and Event Controller and additional modules for the sample preparation (Figure 1).

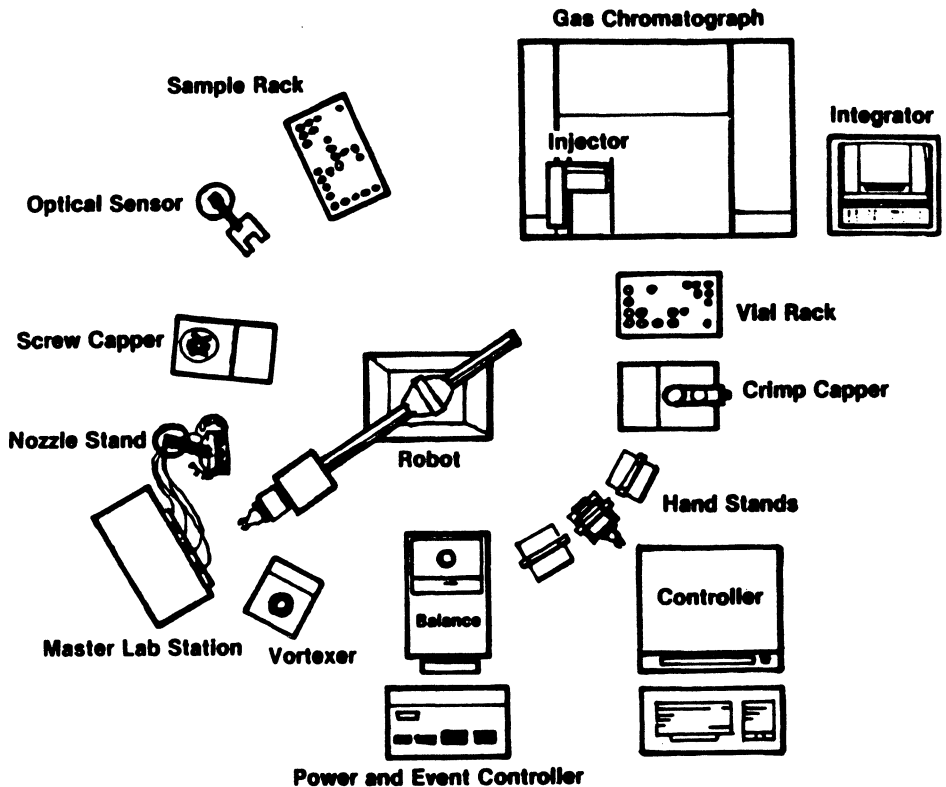


Figure 1

The Hewlett-Packard gas chromatography system includes an HP 5890 Gas Chromatograph, HP 7673A Automatic Injector, and an HP 3392A Integrator which communicate with one another over an INET (Instrument Network) communications loop (Figure 2).

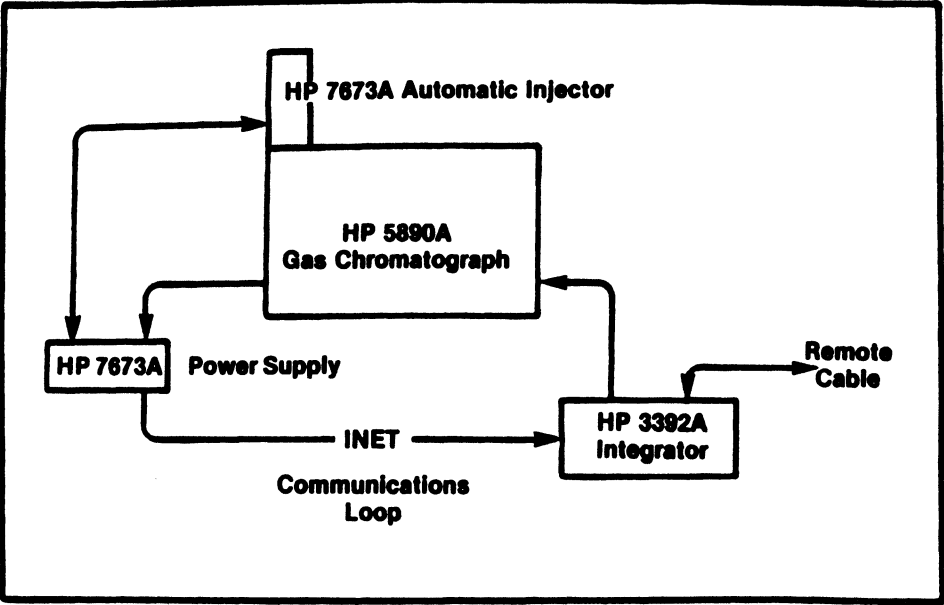


Figure 2

EXAMPLE ONE

The HP 3392A integrator acts as a controller on the INET loop and stores pertinent setpoint and control information about each of the loop devices in files called workfiles. The information in these workfiles can be accessed, and if necessary, edited and transmitted via an RS-232 port on the HP 3392 integrator. The Zymark controller also uses an RS-232 module card for communication with other RS-232 devices or computers. However, the software protocols used by the two systems are not compatible and cannot be altered to let the integrator and the Zymark controller communicate directly. An HP 150 personal computer, though, can be set up to act as a translator between the two and also provide archival storage for the data (Figure 3).

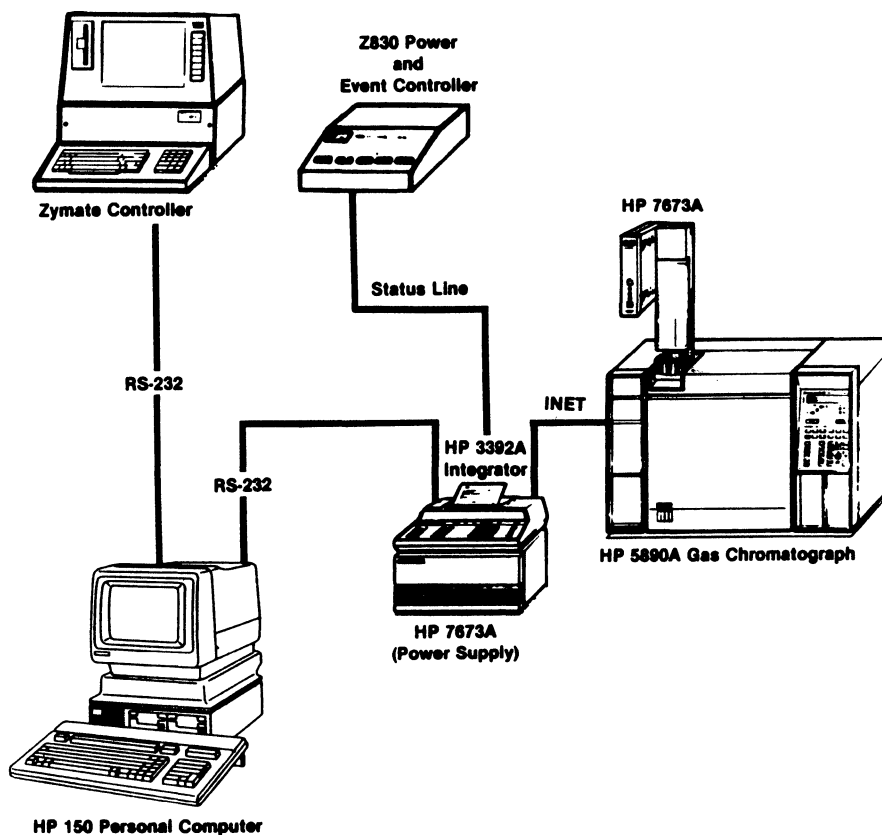


Figure 3

The Zymark system prepares the sample and monitors the status of the GC system (using the Power and event Controller and the "Ready Output" line) to determine if it is ready to accept a sample. The Zymark places the sample in the automatic injector, then sends the sample identification information (weights, volumes, etc.) over the RS-232 interface to the HP 150. The information is not sent until a sample has been placed in the injector to insure that a sample is present prior to a start injection command. A program on the HP 150 then incorporates this information into a workfile that the integrator will use to calculate the amounts of the components of interest in that sample. The computer then sends down a remote start command and waits until the integrator indicates that the run is complete. The results are printed out at the integrator and sent to the HP 150 for archival storage and integration into other database routines. The program then waits for information on the next sample or a message that all samples have been analyzed from the Zymate controller.

EXAMPLE TWO

A more fully automated arrangement takes advantage of the HP 1000-based laboratory automation system in place of the HP 150 as shown in Figure 4.

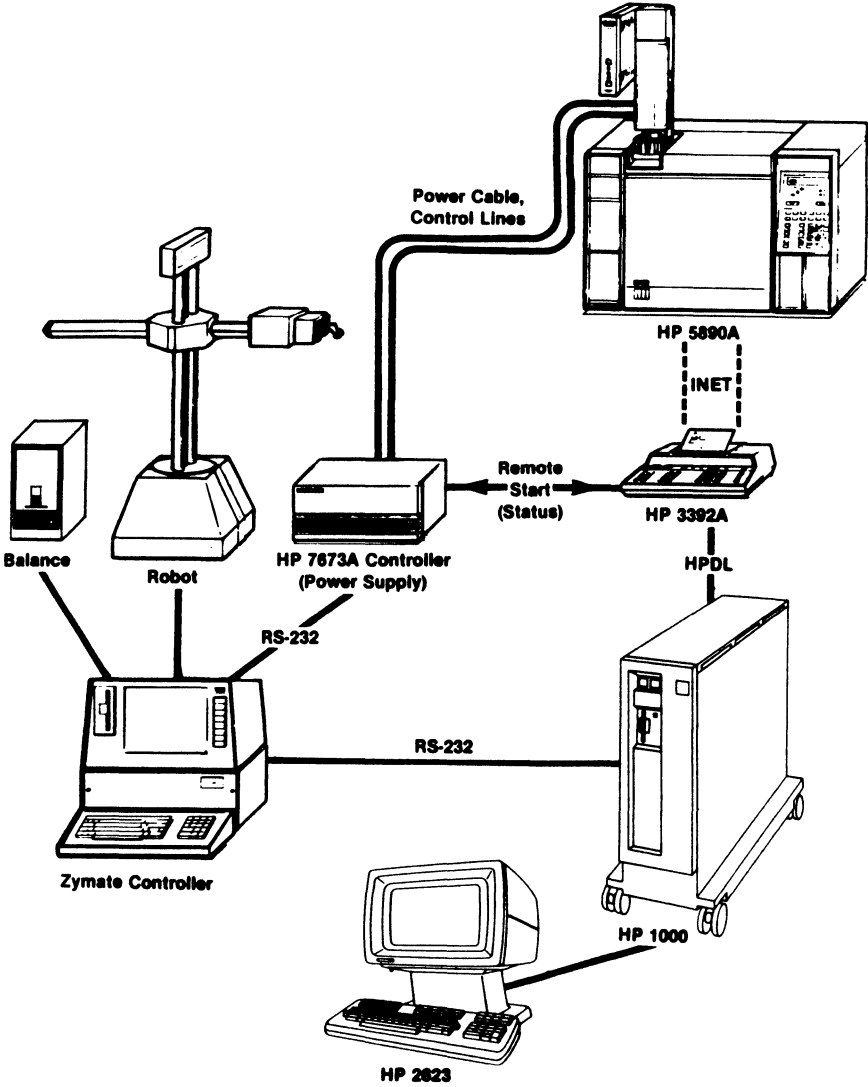
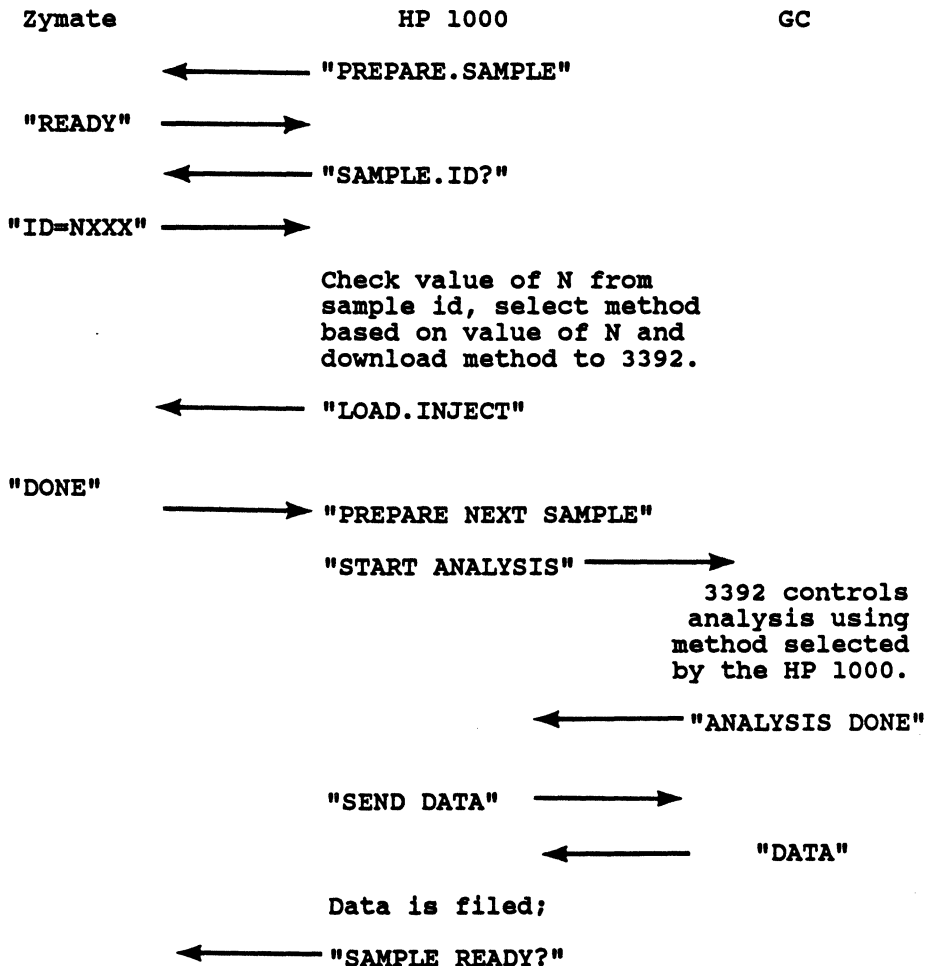


Figure 4

In this case, the HP 1000 initiates the routine by instructing the Zymate to prepare a sample. It then polls the Zymate until the sample is prepared and the sample identification is passed to it. It then instructs the Zymate to load the sample and do the injection. When the Zymate signals that the injection is complete, the HP 1000 instructs it to begin preparing the next sample. The HP 1000 then instructs the GC system to do the analysis, it collects the data, and when that operation is completed, goes back to polling the Zymate to wait for the next sample to be prepared.



Now while the process is repeating itself preparing and analyzing the sample and gathering the data, the 3350 can process the data, store it on a disc for later retrieval, prepare plots and reports, and pass data on to LABSAM/1000 sample management software.

SUMMARY

Based on these examples, you can easily imagine a number of variations to the examples that would tailor configurations and programs to match the processes used in your own laboratories. The most important step in evaluating and planning how to more fully automate and integrate your sample preparation, analysis, and data collection is to first outline, in as much detail as possible, what is required of the combined systems, in terms of answering the laboratory needs. The answer to these needs will include some combination of control, status, error and data lines. The resulting configuration will be dependent upon the equipment involved, the time and funds available to implement it, and the expertise of the people defining and designing the system. Interfacing options will become clearer as robotic systems establish themselves as an integral component of the automated bench.

References:

1. Knipe, Charles R., "The Automated Bench: How Does Your Robot Communicate with Analytical Instruments?"

Performance Analysis and Enhancements for a Vehicle Electrical Test System

David W. Vickers
Stephen E. Novosad

Southwest Research Institute
6220 Culebra Road
P. O. Drawer 28510
San Antonio, TX 78284

INTRODUCTION

Modern luxury automobiles feature many electrical and electronic components. Those components are varied and complex. Some luxury vehicles have several computers as integral parts of their electronic systems. The increased complexity of the electronic systems in vehicles has made the use of more capable test systems necessary to ensure correct assembly and functionality of the vehicles being manufactured. A vehicle Electrical Test System (ETS) has been developed to facilitate the testing of vehicle electrical and electronic systems. Major portions of the test system software were written by Southwest Research Institute (SwRI).

The ETS consists of a network with three kinds of nodes. Development nodes allow test designers to develop vehicle tests. Development nodes also supply most of the support capabilities for the ETS including tester system configuration data, downloading of test sequences to tester nodes, and archiving of all the components and descriptions needed for the development and execution of tests. Gateway nodes connect the tester network to the plant vehicle database computers, to the plant quality control computers and to the development nodes. Tester nodes are used to identify vehicles and run the appropriate tests on them.

This paper will briefly describe the ETS and then focus on several techniques used to enhance tester node performance during tests execution.

DESCRIPTION OF ETS

The development nodes are based on HP A900s with 3 Mbytes of memory and at least 468 Mbytes of disk storage on each node. The development nodes also include the many peripherals necessary to develop tests, such as printers, plotters, graphics tablets, color CRT terminals and, in some instances, tape drives. The development node software consists of several forms-based editors, a test group compiler, a test sequence linker and a set of general utilities. The test designer first

describes the test system, its hardware and its connections to the vehicle. The test designer then describes the tests to be performed in logical blocks called groups. The compiler is used to generate intermediate files containing the test and hardware information for the test groups. The test designer then describes the relationship between the test groups in an entire vehicle test called a test sequence. The linker, provided with the sequence description and the group intermediate files, is then used to link the sequence. When the tests have been linked they can be downloaded to tester nodes to be executed.

The gateway nodes are HP A600s with 1 Mbyte of memory and small disk drives. The gateway nodes are used to communicate between the tester nodes and a vehicle database external to the ETS to retrieve vehicle-specific information necessary to test a vehicle. The database contains the option content for each vehicle. The option content affects the test requirements for the vehicle. The gateway nodes are also used to communicate quality and historical information to other systems in the plant. In addition, the gateway systems have the capability to report the results of recent tests completed by the tester nodes.

The tester nodes consist of Micro-26s. A Micro-26 consists of an A600 computer with an integral Winchester disk and 3 1/4-inch floppies in a 19-inch rack mountable cabinet. The A600s contain 2 Mbytes of memory each and the internal Winchesters contain 55 Mbytes. The tester nodes also include an Operator Interface Unit (OIU) which has a touch sensitive color graphics CRT screen. Other custom hardware is added to the tester nodes as necessary to interface to the vehicle. The tester software provides initialization, operator logon, task selection, vehicle identification and test execution.

Figure 1 is a diagram of a specific plant network. All of the specially designed interconnections between nodes on the plant floor use fiber optic DS connections.

SOFTWARE ENVIRONMENT DURING TESTS

During test execution, several programs are executing concurrently in the tester nodes. The Vehicle Test Execution Supervisor (VTES) controls overall execution of the tests. Several programs are used to interact with the operator through the OIU, with data going to or from those programs through Class I/O. Other programs are used to send data to the vehicle or to receive data from the vehicle. Up to 10 Monitor and Alarm programs may be spawned by VTES to control parallel portions of vehicle tests. Figure 2 gives an overview of the important programs which execute during vehicle testing.

For some I/O, the test control programs (VTES, Monitors and Alarms) read and write directly to or from I/O cards. For other I/O, specifically I/O to the OIU and I/O to the serial interface to the vehicle, the I/O

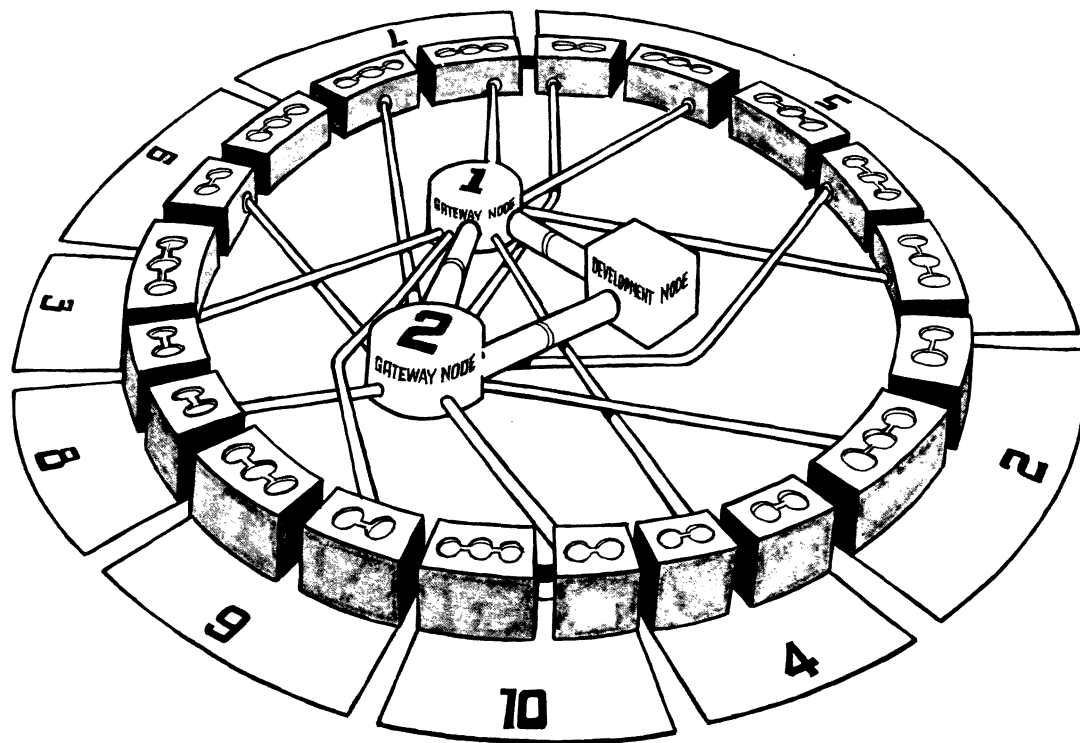


FIGURE 1
DEVELOPMENT GATEWAY AND TESTER NODES SPECIFIC NETWORK

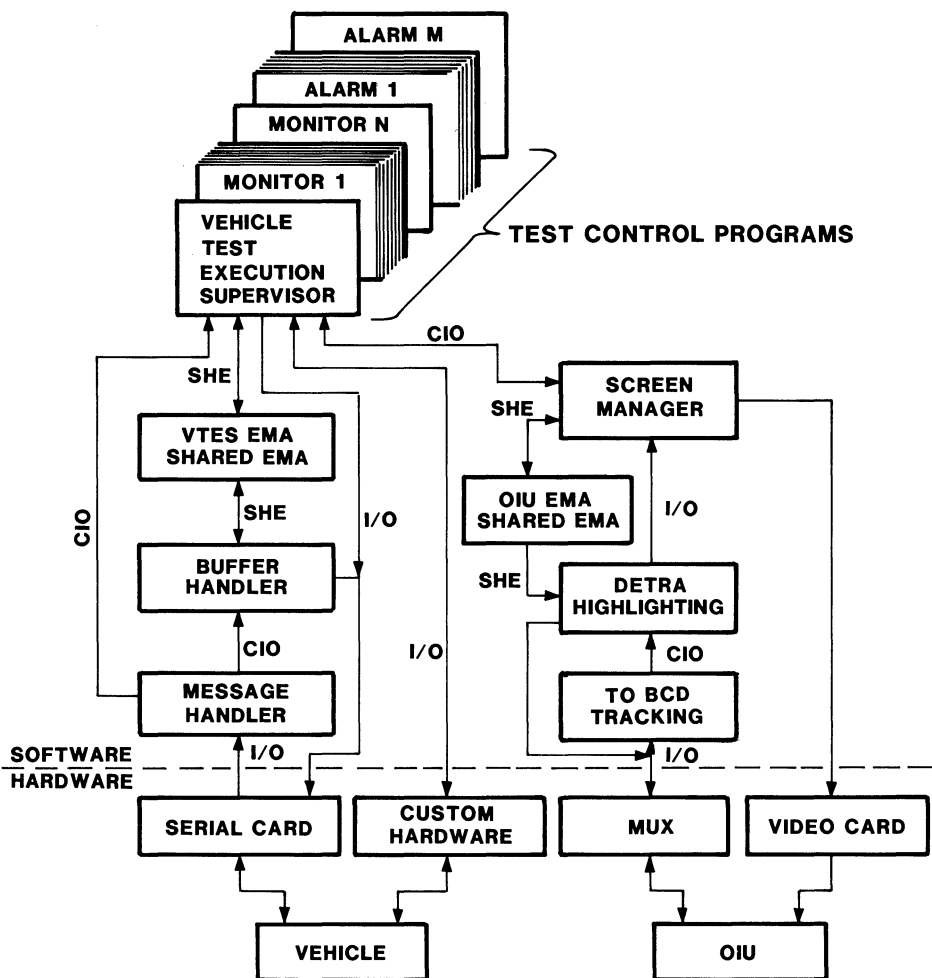


FIGURE 2
MAJOR PROGRAMS DATA FLOW PATHS
DURING TEST EXECUTION

is routed through Class I/O to other programs where it is processed and placed in memory for use by the test control programs.

When a byte from a message on the Serial Diagnostic Link (SDL) interface to the vehicle is to be used by a test, then the test control program needing the byte sets up information in tables in the shared Extended Memory Area (EMA) used by VTES (TESEMA). The test control program also initiates collection of the message containing that byte, if necessary, and then continues to other tests, occasionally checking to see if the data has been collected. When the message is received, it is retrieved by the Message Handler and passed through Class I/O from the card to the Buffer Handler. The Buffer Handler processes the information in the message and puts it in the tables in TESEMA. The next time the test control program checks, it retrieves the data from the tables to complete the test.

When a test control program needs to put a message on the CRT screen of the OIU, it sends a request to the Screen Manager through Class I/O. The Screen Manager then puts the message onto the screen. When the operator touches the touch screen, the tracking program places a cross hair where the screen is touched. When the touch is removed, the tracking program sends the coordinates of the touch to the highlighting program which checks to see if the touched point is in an activated area. If it is, the highlighting program highlights the area and sends an indication of the touch back to the test control program which activated that area on the screen.

PERFORMANCE IMPROVEMENTS DURING DEVELOPMENT

During initial development, programs were assigned default characteristics. Priorities defaulted to 99. Programs defaulted to Code and Data Separation (CDS) and the linker was allowed to segment programs. Once individual programs were ready for integration, work could begin on tuning the system for higher performance.

The relationship between the priorities of the programs and the two fences was one of the first variables tuned. The two fences are the background fence, set at 30, and the time slicing fence, set at 50. Programs with lower priorities (higher numbers) than the background fence are swapped out rather than those with higher priorities. A program with a priority higher than the time slicing fence is executed exclusively as long as it is CPU bound, but those programs with lower priority are time sliced.

Since some DS 1000 programs use real-time priorities and DS 1000 was necessary, we did not change the boundaries from the default values. Also, no programs were put at a higher priority than the real-time DS 1000 programs. The next higher programs, those between 30 and 51, were

background programs but were not time slicing programs. The programs assigned to priorities in this range were those doing the data acquisition, the initial analysis and the interaction with the OIU screen. These programs, since they provided new data to the tests, needed to execute as rapidly as possible. They also had to execute in a short enough time to allow the other programs to continue. These programs were therefore assigned priorities in the 30 to 51 range.

Test control programs differed from data acquisition programs in that test control programs, by definition, must time share so that they can operate concurrently. The test control programs were, therefore, given a priority of 51, immediately below the time sharing fence. The other programs, not needing to operate during actual test execution, were given the default priority of 99.

After the programs in the system had reasonable priorities, the next area of investigation centered on the swapping of programs. Several avenues were utilized in the effort to reduce swapping. Programs were segmented by hand, made into shared code programs and/or assigned to partitions.

Using the WH command and writes in the code showed that, during the execution of vehicle tests with many test control programs, memory was being fragmented and only three or four test control programs could simultaneously reside in memory. Programs which were required during tests, especially small ones which tend to make fragmenting worse, were assigned to partitions. Programs not required during actual testing were assigned to the same partitions as larger programs which were required during tests to ensure that the ones not required were swapped out during testing.

The VTES and Monitor programs (the Monitor program executes Alarms as well as Monitors) were each hand segmented. The subroutines in VTES were segregated by use, since some were used only during part of the vehicle tests and others were used only at some test sites or during abnormal circumstances.

VTES was initially linked as three 31-page segments. Each segment contained a random set of subroutines. The program was hand segmented using the Link NS (New Segment) command in the Link input file, and the subroutines were rearranged so that six of ten segments were in memory at one time using the Link CD (CoDe) command. Overall, some memory was saved and almost no swapping occurred.

The Monitor program was also initially linked as three 31-page segments and a copy of each segment was made each time the monitor program was cloned. The code in the Monitor program is, in reality, only about 63 or 64 pages long. The linker used 91 pages of physical memory because it used the default segment size of 31 pages. Using the Link NS command and specifying a size, the linker can be forced to break the program up

into three nearly equal length segments so that it only takes up about 67 pages of physical memory.

The largest saving in physical memory space, however, was made by making the Monitor program into a Shared code Program (SP). Since the program was already a CDS program, adding the SP command to the Link command file was all that was required to make it a shared program. By making the Monitor program a shared program, only one copy of the code segment was kept in memory and memory space was saved.

POST-DEVELOPMENT ENHANCEMENTS

The techniques used above provided a useable system; however, as line speeds increased in the plant, the vehicle tests needed to run faster. During integration, tradeoffs were made to increase the performance of some innermost loop routines at the cost of readability or modularity. After integration, two techniques were applied to provide additional performance enhancements. They were: (1) using large file reads, and (2) changing the system time quantum.

The techniques used to determine which subroutines were worth working on and finding the information required to improve those subroutines were dictated by the characteristics of the tester and development systems. All tester software was developed on one of the development nodes. The tester nodes did not contain the source code for the tester software. The Symbolic Debugger was therefore of little use. When a study of the runtime data produced by the tester software was not sufficient to solve a problem, the normal debug method was to put write statements into the code to find and fix the problem.

The first technique used to measure the performance of various portions of the software was the insertion of write statements. In order to minimize affect on execution, the timing data was taken using a cross load from the system time count, \$TIME. The timing data was put into an array and printed after execution was complete. When more complex and less intrusive techniques were needed, an HP-64000 system with an interface to the A-600 backplane was used. For those programs which were not shared code and were all in memory at one time, the address lines of the backplane gave excellent statistical information on which parts of code were being used. The major drawback was the limited number of data points over which data could be taken. This limitation could be somewhat overcome in two ways. The first was to take statistical data over smaller portions of the program, and the second was to take data in a repeat mode and estimate the averages as the numbers on the screen changed.

Using the logic analyzer, data also was obtained from shared code programs; however, that data was less easily used since it came from more than one program executing at one time. Data could be obtained from

programs not normally in memory by putting all segments into memory. Unfortunately, this method provided less information about actual execution behavior, since the swapping of segments did not occur. The logic analyzer also provided statistical data regarding execution of system routines, but not about DMA-type activity such as data transfers from disk. Using the data from the write statements and the logic analyzer, and general knowledge of HP-1000 systems and the TES in particular, two major and several minor changes were made.

Figure 3 gives an overview of the elements of the tests as seen by the test execution software. A particular sequence is executed based on the vehicle to be tested. The sequence consists of partial tests called groups. These are executed serially by VTES which also spawns the Monitors and Alarms that execute in parallel with VTES. The basic structure of Groups, Monitors and Alarms is the same. The data that VTES and the clones of the Monitor program need to execute the Groups, Monitors and Alarms is provided to the tester in type 2 files (files consisting of records of a constant, predefinable length) with various short record lengths. Since the files were being read one record at a time, the actual reading of the Group-type files was taking a significant part of the test time.

One way to decrease the time required to read the files was to force the files to type 1 (record length equals 128) using the F option during the FmpOpen call and then to read the files into a large buffer. There is, however, no room for a large buffer in the data segment of either VTES or the Monitor program. A separate program (Group Read) was therefore written which has room for a large buffer in the data segment. The new program has access to TESEMA, where the data from the files must be stored. Only two parameters must be passed to Group Read to put the data in the correct places in TESEMA. These parameters are passed to Group Read through the run string. Some data must be passed directly from Group Read to the calling test control program. That data is passed back via an EXEC 14 string passage call using a buffer reserved by the calling test control program which scheduled Group Read. After some experimentation, the tests were found to execute faster if the EXEC calls scheduling Group Reads were queued. The EXEC calls, of course, must be with "wait," since a return buffer is expected. The scheduling EXEC call is therefore an EXEC 23 queue schedule with "wait."

In order to get the information into the correct place in EMA using the type 1 reads, more calculations must be made. A call to Group Read results in the following actions by Group Read. The first read gets the code for the Group or Monitor to be executed. The second read provides the data necessary to calculate the position and length of all the data for the Group or Monitor. The next two reads get all of the data. Then, using the information from the second read, all of the data is separated and put into the correct places in EMA.

DEFINITIONS:

SEQUENCE =: [SEQUENCE STATEMENT] *, END SEQUENCE
 SEQUENCE STATEMENT \in { SPAWN MONITOR/ALARM,
 SEQUENCE CONDITIONAL,
 GROUP,
 KILL MONITOR/ALARM }

GROUP =: [GROUP STATEMENT] *, END GROUP, FAULT MAP
 GROUP STATEMENT \in { GROUP CONDITIONAL,
 PRIMITIVE FUNCTION }

MONITOR =: [GROUP STATEMENT] *, END GROUP
 ALARM = [GROUP STATEMENT, ALARM CONDITION] *, END GROUP

EXAMPLE:

SEQUENCE A		
SEQUENCE CONDITIONAL		
SPAWN MONITOR	-----	MONITOR
GROUP B	-----	GROUP B
SEQUENCE CONDITIONAL	PRIMITIVE D	GROUP CONDITIONAL
SPAWN ALARM	PRIMITIVE E	PRIMITIVE
GROUP C	PRIMITIVE F	GROUP CONDITIONAL
KILL MONITOR	GROUP CONDITIONAL	PRIMITIVE
KILL ALARM	PRIMITIVE G	PRIMITIVE
END SEQUENCE	PRIMITIVE H	END GROUP
	END GROUP	
	FAULT MAP	

FIGURE 3

During actual execution of tests, especially at those sites where the serial line is the only connection to the vehicle, much of the execution time is spent polling the primitive functions to determine if they have been completed. This process is normally compute-bound since it consists primarily of checking tables in memory. Since the test control programs have priorities above the time swapping fence, they execute until the quantum is over. During tests, a test control program will sometimes loop, delaying testing until a parallel test control program has determined that a stable condition exists in the vehicle. Therefore, some of the time that control programs are executing is not being used effectively. To be most effective, programs should execute their loop once to check for data, but should not continue to execute if there is no new data. The system quantum determines the amount of time a program is executed before another program is executed. Experiments were made to determine the effect on the total test time of changing the system time quantum. The raw results of those experiments are shown in Tables 1 through 3.

One of the most indicative experiments was the first one. The data from Table 1 is plotted in Figure 4. The decrease in overall time and in the standard deviation of the time as the time quantum is changed from the 300-millisecond default to 100 milliseconds is obvious. Below 100 milliseconds, although there are still some shorter tests, there are also some longer tests. The data from Tables 2 and 3, together with data from Table 1, indicate that the optimum time quantum is between 80 and 110 milliseconds. Since the time quantum is only variable in 10's of milliseconds, 100 milliseconds was chosen as the time quantum.

The data gathered during the experiments could not always be placed on a simple smooth curve. There are several reasons for the variations. The amount of time taken by a particular test was dependent on some external variables. For example, if the test was run on a vehicle with a cold engine, the test took much longer to complete. The number of samples at each quantum may not have been enough to provide a statistically significant sample. Small variations and inconsistencies in the operation of the vehicle during the tests could, at times, have had a significant effect on the time. For example, the test that took 184 seconds to complete with the time quantum set to 250 milliseconds was almost certainly the result of some vehicle condition. As the time quantum was reduced below 100, the dispersion of the times increased significantly. Overall, the data did indicate the basic trends involved in changing the time quantum and provided sufficient information to determine a suitable value for it.

CONCLUSION

There are many different methods which can be used to increase the performance of a system. The methods used during the development of a system may differ in some ways from those used later, but they are in many

TABLE 1
TIMES (IN SECONDS) FOR 10 TESTS AT 50-MILLISECOND INTERVALS

		Time Quantum (Milliseconds)					
		<u>50</u>	<u>100</u>	<u>150</u>	<u>200</u>	<u>250</u>	<u>300</u>
T E S T N U M B E R	1	143	144	147	157	168	172
	2	146	142	150	146	184	178
	3	142	138	147	153	165	168
	4	148	136	151	156	156	159
	5	155	143	149	149	157	156
	6	136	138	150	156	160	167
	7	138	135	150	159	162	162
	8	146	139	147	151	159	174
	9	141	147	151	160	171	162
	10	133	142	144	155	151	160
Max Time		155	147	151	160	184	178
Min Time		133	135	144	146	151	156
Avg Time		142.8	140.4	148.6	154.2	163.3	165.8

TABLE 2
TIMES (IN SECONDS) FOR 10 TESTS AT INTERVALS
BETWEEN 50 AND 110 MILLISECONDS

		Time Quantum (Milliseconds)			
		50	70	80	110
T E S T N U M B E R	1	140	150	141	140
	2	144	133	156	147
	3	140	137	143	148
	4	138	148	146	144
	5	142	150	143	140
	6	149	143	137	144
	7	141	149	154	150
	8	149	139	148	143
	9	147	136	168	148
	10	149	134	150	145
Max Time		149	150	168	150
Min Time		138	133	137	140
Avg Time		143.9	141.9	148.3	143.2

TABLE 3
TIMES (IN SECONDS) FOR 10 TESTS AT 50 AND 100
MILLISECONDS WITH AND WITHOUT OPERATOR ACTIONS

		<u>Time Quantum (Milliseconds)</u>			
		<u>50</u>	<u>50A</u>	<u>100</u>	<u>100A</u>
T E S T N U M B E R	1	164	135	145	137
	2	139	134	148	148
	3	140	132	138	129
	4	141	134	148	135
	5	144	125	144	135
	6	148	146	143	131
	7	155	129	148	145
	8	146	130	143	142
	9	146	130	139	129
	10	146	131	146	141
Max Time		164	146	148	148
Min Time		139	125	138	129
Avg Time		146.9	131.6	144.2	137.2

⊗ ANOMILY

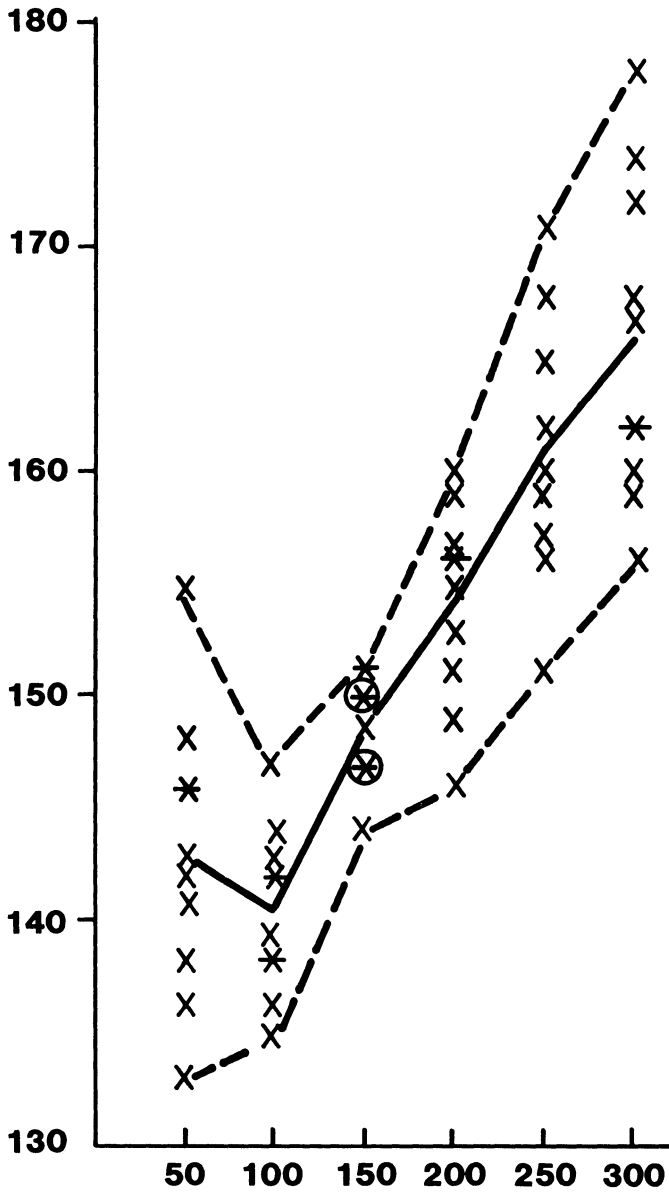


FIGURE 4
SCATTER PLOT FOR 50 MILLISECOND INTERVALS

respects similar. Enhancements, both during and after development, consist largely of two related actions. The first is to determine what system tasks are taking significant amounts of time. The second is to tune the use of the existing system to optimize or eliminate those system tasks. Attempts to optimize all parts of a large system may be inappropriate, but optimizing the most critical or the most time consuming portions of a system can be very beneficial.

Using the Touchscreen Features of the HP150 in Application Programs

Michael D. Green
Kenneth L. Kueny
Department of Aerospace Engineering
University of Maryland
College Park, MD 20742

I. Introduction

Many devices are available for manual input of data into computers. Keypads, barcode readers, the mouse, joysticks and touchscreens are a few examples. Each device has advantages and disadvantages in ease of use, resolution, hardware complexity, error rejection, robustness, and size. The Hewlett Packard 150 personal computer incorporates a touchscreen into its display. The main advantage of the touchscreen is that the input mechanism is combined with the display and no additional hardware is required. Many applications are available for the HP150 personal computer which utilize its touchscreen features. This paper will describe how to use the touchscreen as an input device to HP1000 application programs.

II. Hardware Overview

The touchscreen consists of a grid of 21 by 14 pairs of infrared emitters and photo-detectors along opposite sides of the screen bezel [Ref. 1]. A keyboard and touchscreen controller scans the emitters and detectors checking for a blocked pair. The data is averaged if two adjacent pairs are interrupted and thus the resolution is greater than the number of pairs used. A valid hit is reported to the controller as one of 27 rows and one of 41 columns.

III. Software

The touchscreen capabilities are accessed through sequences of escape codes [Ref. 2,3]. The codes are defined below:

A. Reporting Mode

There are two modes of detecting 'touches'; field reporting and row-column reporting. Field reporting defines areas of the screen as fields and also establishes the response of

the fields to touch. Field mode is useful for menu driven applications. Row-column reporting senses the row and column address of a touch and is useful for cursor positioning.

The default mode for touchscreen reporting is off, touches are not reported. The following escape sequence sets up the touch field reporting modes.

```
ESC - z <smode> n [<tmode> M]
```

where

<smode> defines the screen mode.

- 0 = Turn off all reporting without deleting any fields.
- 1 = Enable row-column reporting only. Disable field reporting, row-column address returned.
- 2 = Enable touch-field reporting only. Reports from defined fields.
- 3 = Enable row-column and field reporting. Defined fields are returned if touched, other areas of the screen are returned as row-column address.
- 4 = Toggle touchscreen on/off.

<tmode> defines the touch mode for row-column reporting. Tmode is required for row-column sensing but is optional for field sensing. Possible values for tmode are:

- 1 = Report on touch only.
- 2 = Report on release only.
- 3 = Report on touch and release.

B. Defining a touch field

If touch field reporting is chosen the following escape sequence defines the field.

```
ESC - z g <R> r <C> c <curs> p <beep> b <off> e  
      <on> f <attr> a <mode> m <buff-len> L  
      <buf>
```

where

- <R> = The beginning and ending rows for the field.(0 to 47) Entered as <start-row,end-row> and end-row must be greater than or equal to start-row.
- <C> = The beginning and ending columns for the field.(0 to 79). Entered as <start-column,end-column>, again end-column must be greater than or equal to start-column.
- ~curs> = The position of the alpha cursor when the field is touched. If <curs> is omitted the cursor is not positioned on touch. Values for <curs> are
- 0 = The cursor is not positioned.
 - 1 = The cursor is positioned at the upper left corner of the field.
- <beep> = The sound made when a field is touched. If <beep> is omitted no beep occurs. Values for <beep> are
- 0 = No sound is made when field is touched.
 - 1 = The system beeps when field is touched.
- <off> = The enhancement displayed when the field is off (not touched). Values are displayed in the enhancement table shown below. If <off> is omitted the default value of 10 (half-bright, inverse) is assumed.
- <on> = The enhancement displayed when the field is on (touched). Again, possible values are displayed in the table below. If <on> is omitted the default value of 2 (inverse) is assumed.

Touch Enhancement Table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Half-bright									X	X	X	X	X	X	X	X
Underline					X	X	X	X					X	X	X	X
Inverse			X	X			X	X			X	X			X	X
Blinking	X			X		X		X		X		X		X		X

<attr> = The type of field to be defined.

- 1 = ASCII field
- 3 = Toggle field
- 4 = Normal field

The next section describes the types of fields available.

<mode> = The sensing mode for this field. Values for mode are :

- 1 = Report on touch.
- 2 = Report on release.
- 3 = Report and release.

<buf-len> = The length of the character string associated with this field.

<buf> = The character string associated with the field. Buf can be up to 80 characters for ASCII fields and must be 2 characters for Toggle and Normal fields.

C. Types of touch fields

1 = ASCII field. A buffer of 80 characters is associated with the field. When the field is touched the buffer is sent to the application.

3 = Toggle field. When the field is first touched it is toggled on, when touched again it is toggled off. When touched it returns the following escape sequence:

ESC - z <buf> <state> Q

Where

<buf> is a two character buffer as specified at the end of the main escape sequence, thus <buf-len> needs be defined as 2. <Buf> can be used to identify which field has been activated and is returned in lower case.

<state> returns one of two values:
1 = toggle field turned ON
2 = toggle field turned OFF

4 = Normal field. Similar to ASCII fields but only two characters are associated with <buf>. The following escape sequence is returned to the application program.

ESC - z <buf> <type> Q

Where

<buf> = two character buffer
returned, again in lower-case.

<type> = 5 when normal field is
touched
= 6 when normal field is
released

NOTE: A toggle field has two states, either ON or OFF whereas a normal field is ON only while being touched.

D. Row-column reporting

When row-column reporting is activated the following escape sequence is returned when the screen is touched.

ESC - z <row> x <col> y <type> Q

Where

<row> = two-digit integer specifying the row.
(0 to 47)

<column> = two-digit integer specifying the
 column. (0 to 79)
<type> = 3 when row-column is touched.
 = 4 when row-column is released.

E. Deleting Touch Fields

When defined the touch fields remain in operation within the two page alpha memory unless explicitly disabled. Even if they are scrolled off the screen they still remain in effect and will be active if scrolled down again. The following escape sequence removes a single touch field.

ESC - z d <row> r <col> C

Where <row> and <col> define any coordinate within the touch field. No action is taken if no field is defined at this point.

The following escape sequence deletes all touch fields.

ESC - z D

F. Resetting Touch Fields

Resetting the touchscreen turns all fields OFF but does not affect touch sensing or reporting. The following escape sequence resets the touch fields.

ESC - z J

IV. Sample Program

The following is a sample program developed to demonstrate the use of the touchscreen principles described above. There are two main applications for user defined touch fields: data entry and command entry and the sample program demonstrates both. A menu driven touchscreen program requires scroll suppression and a pending read following the touch field definition. Without scroll suppression the touch field will scroll off the screen. The touch field response is not detected unless the driver program has a pending read.

The main program is just a simple driver, however, the subroutine FIELD could be used in any application. It generates touch fields that are 3 rows high and 18 columns wide.

The output from the sample program appears as follows. Touching the displayed values generates a request to enter a new value for the parameter which is then updated. Touching the STOP field stops the program, and then the touch fields are disabled. The field which has been touched is determined by checking the value of the returned character buffer.

The diagram shows a rectangular frame containing a 3x3 grid of rectangular boxes. Each box contains a text label. A hand is shown on the left side, with the index finger touching the top-left box. Below the grid is a single rectangular box labeled 'STOP'. At the bottom of the frame is a text prompt followed by a blank line for input.

V (1) = 1.	V (2) = 4.	V (3) = 9.
V (4) = 16.	V (5) = 25.	V (6) = 36.
V (7) = 49.	V (8) = 54.	V (9) = 81.

STOP

Enter new V (1): _

PROGRAM TOUCH

CHARACTER*10 RESP
INTEGER R,C
REAL V(9),X

```
*****
*
*
*           General Purpose Touch Field Program
*
*           Department of Aerospace Engineering
*
*           University of Maryland
*
*           College Park, Maryland
*
*           8 January 1986
*
*****
```

c Written by: Michael D. Green & Kenneth L. Kueny

c Generate Dummy Data for display.

```
DO I=1,9
  V(I)=I*I
END DO
```

c Home cursor and clear display

```
5  WRITE(1,*) CHAR(27),'H',CHAR(27),'J'
```

```
c Call FIELD Subroutine (Row,Column,Label,Response,Value)
c   Row      => defines the top row of the touch field
c   Column   => defines the left hand column of the touch
c               field
c   Label     => 6 character string defining touch field
c               label
c   Response  => 10 character string defining touch field
c               response
c   Value     => value to be displayed in touch field
```

```
CALL FIELD(1,1, 'V(1)= ', 'New V(1)= ',V(1))
CALL FIELD(1,30,'V(2)= ', 'New V(2)= ',V(2))
```

```

CALL FIELD(1,60,'V(3)= ','New V(3)= ',V(3))
CALL FIELD(5,1, 'V(4)= ','New V(4)= ',V(4))
CALL FIELD(5,30,'V(5)= ','New V(5)= ',V(5))
CALL FIELD(5,60,'V(6)= ','New V(6)= ',V(6))
CALL FIELD(9,1, 'V(7)= ','New V(7)= ',V(7))
CALL FIELD(9,30,'V(8)= ','New V(8)= ',V(8))
CALL FIELD(9,60,'V(9)= ','New V(9)= ',V(9))
CALL FIELD(13,30,'STOP ','Hit return',X)

```

c Read touch field response

```

10  FORMAT (A10,F4.4)
    READ (1,10) RESP,X                      ! Pending read
    WRITE(1,*) CHAR(27),'A',CHAR(27),'K'    ! Scroll off

    IF (RESP.EQ.'Hit return') THEN          ! Stop selected
        GOTO 99                             ! turn off fields
    ELSE
        IF (RESP.EQ.'New V(1)= ') V(1)=X
        IF (RESP.EQ.'New V(2)= ') V(2)=X
        IF (RESP.EQ.'New V(3)= ') V(3)=X
        IF (RESP.EQ.'New V(4)= ') V(4)=X
        IF (RESP.EQ.'New V(5)= ') V(5)=X
        IF (RESP.EQ.'New V(6)= ') V(6)=X
        IF (RESP.EQ.'New V(7)= ') V(7)=X
        IF (RESP.EQ.'New V(8)= ') V(8)=X
        IF (RESP.EQ.'New V(9)= ') V(9)=X
        GOTO 5                               ! Redraw screen
    ENDIF

```

c Main body of program would go here. Other IF statements
c can be used to determine action to be taken for a given
c response.

c eg. programatic scheduling

c Deactivate all touch fields

```

99  WRITE(1,*) CHAR(27),'-zD'

    STOP
    END

```

c -----

```

SUBROUTINE FIELD(R,C,LABEL,RESP,Z)

```

```

CHARACTER*6 LABEL

```

```

CHARACTER*10 RESP
CHARACTER*1 E,CR
REAL Z
INTEGER R,C,H,W

```

```

c      H=2      ! Row height of touch field minus one (Can be
              modified)
c      W=17     ! Column width of touch field minus one (Can be
              modified)

```

```

E=CHAR(27)      ! Escape character definition
CR=CHAR(13)     ! Carriage return character definition

```

c Row bound error message

```

IF (R.GT.23-H.OR.R.LT.1) THEN
  WRITE(1,*) E,'-zD'
  WRITE(1,*) '>>> ERROR <<<'
  WRITE(1,*) 'Row parameter for ',label,' is ',R,'.'
  WRITE(1,*) 'Row parameter range is 1 to ',23-H,'.'
  STOP
END IF

```

c Column bound error message

```

IF (C.GT.79-W.OR.C.LT.1) THEN
  WRITE(1,*) E,'-zD'
  WRITE(1,*) '>>> ERROR <<<'
  WRITE(1,*) 'Column parameter for ',label,'
$      is ',C,'.'
  WRITE(1,*) 'Column parameter range is 1 to ',79
$      -W,'.'
  STOP
END IF

```

c Activate touch field reporting only.

```

WRITE(1,*) E,'-z2N'

```

```

c Define touch field with predefined row and column
c address, no cursor positioning, beep on, half bright
c inverse field when off, inverse when on, ASCII field
c report on touch, buffer length 10 characters, buffer
c predefined as 'resp'.

```

```

WRITE(1,*) E,'-zg',R,',',',',R+H,'r',C,',',',',C+W,'clbl0e2fla
&lm10L',RESP//CR

```

c Position cursor and label touch field

```
IF (LABEL.NE.'STOP ') THEN
  WRITE(1,*) E,'&a',R+1,'r',C+1,'C',LABEL,Z
ELSE
  WRITE(1,*) E,'&a',R+1,'r',C+1,'C',LABEL
END IF
```

c Position cursor at bottom of screen for response

```
WRITE(1,*) E,'&a18r1C'
```

```
RETURN
END
```

c -----

V. References

1. HP 150 Technical reference manual. PN 45625A
2. HP 150 Programmer's reference manual. PN 45435-90002
3. Grow your own touch. Bill Crow. Professional Computing
Dec/Jan 1985 Volume 1, Number 5.

SETKY-GETKY, A KEYED ACCESS SYSTEM FOR THE HP1000

Dorothy Bickham
David Neumann
Chemical Thermodynamics Division
National Bureau of Standards
Gaithersburg, MD 20899

INTRODUCTION

SETKY-GETKY (1) is a public domain keyed access system written for the Hewlett-Packard HP1000 mini-computer*. Its main function is to provide rapid access to free formatted textual or tabular material stored in large data files. It provides a choice among output devices and some user control over the format of the material displayed. Applications where SETKY-GETKY could be used include an online phone directory, a correspondence file, a data base of computer users or vendors, an online help system, and a storage and retrieval system for tabular data.

The basic characteristic of a keyed or indexed access system is that, in order to access data in a data file, a separate, smaller, more rapidly searchable file is created containing only the keys to and the locations of the records in the original data file. Such a file is called an index or key file (2). It provides a means of rapid retrieval, via direct access, of records from the data file. In the SETKY-GETKY system, the keys or keywords are defined by the user and serve to identify, for future retrieval, groups of records in the data file.

The GETKY program performs the data retrieval for a requested keyword when invoked by the user as follows:

```
CI> GETKY, <keyword>, <keyfile>
```

In the above, and in other examples below, "CI>" is the command interpreter prompt issued by the HP operating system. The form of the GETKY runstring allows for other options and parameters, some of which

* Certain commercial equipment including computer software is identified in this paper in order to adequately specify procedures, experiments, and techniques used. Such identification does not imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the materials, equipment, or software identified are necessarily the best available for the purpose.

are shown below. A complete description can be found in the manual^(1,3) for the system. The SETKY program generates the key file used by GETKY to retrieve the data and is invoked as follows:

```
CI> SETKY, <datafile>
```

Again, details on the SETKY runstring are given in the manual^(1,3).

To implement the SETKY-GETKY system, the user modifies an existing data file by including SETKY commands indicating which groups of data records are to be associated with which keywords. Keywords are ASCII character strings, up to 24 characters in length. Depending on the application, keywords might be author names, program names, abstract numbers, or chemical substance formulas. The data itself can be textual or tabular information, a transfer command, or a command to run a program.

After the data file has been organized, the SETKY program is run to generate the key file used by GETKY. GETKY can then be used to access and display data associated with any keyword. In its interactive mode, GETKY will display on the terminal screen an alphabetized list of keywords that bracket a requested keyword when the keyword itself is not in the key file. The user may then select one of the displayed keywords or move forward or backward through a list of keywords. SETKY-GETKY also provides an automatic facility for transferring from one key file to another.

SETKY and GETKY were modeled after the programs, GENIX⁽⁴⁾ and CMD⁽⁴⁾, respectively, which Hewlett-Packard no longer supports on the HP1000 model A900. GENIX, available under RTE-6⁽⁴⁾, is a utility program which creates index files for use by the HP HELP⁽⁴⁾ and CMD utilities. There are many similarities between SETKY and GENIX and between GETKY and CMD. Both pairs of programs, SETKY-GETKY and GENIX-CMD, provide for the retrieval of free formatted text, allow multiple keys to access the same data, provide a facility for blocking the text into subsections for display purposes, allow transfers from one indexed file to another, and display a list of valid keys to choose from if the user enters an invalid one.

There are some differences between the two systems. CMD allows for a partial match on a key; for example, OXY would be recognized as a match for OXYGEN if no other keyword in the key file began with the letters, OXY. On the other hand, SETKY allows duplicate keys, i.e., one key referencing more than one set of data. GETKY also provides some additional features: spooling of output to a printer, output to a disc

file, execution of run commands in the data, and the availability of a programmatic interface.

Three examples are presented below to demonstrate the use of SETKY-GETKY. The first is a simple example of a database of computer users. The second example shows the development of an online help system and user's manual. The last example involves the storage and retrieval of tables of chemical thermodynamic functions. It illustrates how the SETKY-GETKY programmatic interface subroutines: OpenKy, CloseKy, PositionKy, and ReadKy can be invoked from an application program.

USERS INFORMATION SYSTEM

A small database was created using SETKY-GETKY for the storage and retrieval of pertinent information about the users of our HP1000. The data items stored for each user included first and last name, phone number, building and room number, data center, type of terminal or microcomputer, and mux port. This information, along with the required SETKY commands, was stored in one data file, called USERS.IDX. It is a convention of the SETKY-GETKY system that all data files have an extension of IDX. The program SETKY was run as follows:

```
CI> SETKY, USERS.IDX
```

to create the key file, USERS.KEY. Again, by convention, all key files generated by SETKY have a KEY extension.

GETKY is invoked to retrieve data from USERS.IDX via the location information contained in the key file, USERS.KEY. For example, to display the data stored on the user named Neumann, enter

```
CI> GETKY, NEUMANN, USERS.KEY
```

and the following is displayed.

```
David Neumann  
Phone 921-3632  
Bldg. 222 / Room A157  
Chemical Thermodynamics Data Center  
Vectra Microcomputer  
MUX port 63
```

On the other hand, if a user enters

```
CI> GETKY, HP150, USERS.KEY
```

then, there being more than one HP150 keyword in USERS.IDX, the first entry appears followed by a continuation message:

David Smith-Magowan
Phone 921-2108
Bldg. 222 / Room A160
Electrolyte Data Center
HP150 Microcomputer
MUX port 26

NOTE:

Another occurrence of this key has been found.

Hit <CR> to display, A,<CR> to stop

If the user were looking for the data on the HP150 that was just displayed, he could enter an A (for abort) followed by a carriage return to abort GETKY. On the other hand, if he were looking for data on another HP150, he could enter a carriage return to display the next set of data indexed under HP150.

In our users data file we have information on 32 users, plus 4 modems, and a graphics workstation. This information is keyed not only by user name, but also by data center, terminal type, mux port number, and graphics capability of the user's terminal, if any. For example, the data and SETKY commands for David Neumann are entered in the data file, USERS.IDX, as follows:

```
" "  
"NEUMANN  
"SS  
David Neumann  
Phone 921-3632  
Bldg. 222 / Room A157  
Chemical Thermodynamics Data Center  
"CTDC  
Vectra Microcomputer  
"vectra  
"graphics  
MUX port 63  
"lu63  
"XX
```

All the records that begin with a double quote, ", are SETKY commands and will not be displayed with the data. The pair of double quotes, "",

is used to signal SETKY that a key is coming for a new data segment. The next record is the key, in this case, NEUMANN, preceded by a double quote. The "SS and "XX records are used to mark the beginning and end of the data associated with the key, NEUMANN. Within the demarcated data block, the records that start with a double quote are additional keys that reference the same data as the key NEUMANN does.

These additional keys allow the user to look at the information in the data base in a variety of ways. If the system manager wants to refresh his memory as to who is using an HP2621 terminal, he can enter

```
CI> GETKY, HP2621, USERS.KEY
```

If he is working on some graphics software and wants to determine who might be affected by changes, he can enter

```
CI> GETKY, GRAPHICS, USERS.KEY
```

for a display of the users with graphics monitors, their terminal types, and their mux ports. For instance this information is required for a configuration file used by the GRAFIT/1000⁽⁵⁾ software package.

The GETKY program also offers the option of sending the data retrieved to a printer or a disc file, as well as to the screen. To obtain a listing of all the users who belong to the Chemical Thermodynamics Data Center, indexed under CTDC, on the printer, lu 6, simply enter

```
CI> GETKY, CTDC, USERS.KEY, 6
```

and the GETKY program will retrieve all the data and automatically spool it to the printer.

On a day-to-day basis, it seems that the most common use of this users data file is for the retrieval of phone numbers. To make the runstring short and easy to remember, a short, simple Fortran program, called PHONE, was written. Now a user only has to enter

```
CI> PHONE, <username>
```

where username is the name of the user whose phone number is desired and the information in USERS.IDX on the requested user is displayed. The program, PHONE, calls a subroutine named, RuGetKy, which schedules GETKY with the given username as the key and USERS.KEY as the key file.

ONLINE HELP SYSTEM PLUS USER'S MANUAL

A help system provides both online help information and a hardcopy user's manual. This is accomplished through the construction of a single data file that incorporates not only the information on a variety of topics describing programs, tips, procedures, and other information needed by the users, but also SETKY commands. Then, after this data file is processed by SETKY to produce a key file, online help is available with GETKY. By adding TEXED⁽⁶⁾ commands and running the TEXED program, a user's manual is produced from this same data file. This is possible because SETKY-GETKY is completely compatible with TEXED, document processor for the HP1000.

The data file is named HELP.IDX, the name of the default help file of the SETKY-GETKY system. The SETKY commands are inserted into HELP.IDX by using EDIT/1000⁽⁷⁾. The textual information to be retrieved by GETKY for each individual topic is preceded by a "SS record and followed by a "XX record. A descriptive keyword is placed before each "SS record in the required SETKY format. Frequently one or more additional keywords are used to reference the same text. These additional keywords are placed within the body of the text preceded by a double quote to indicate that they are keywords, not part of the text to be retrieved by GETKY. For example, four print programs are described. Each of these print programs is indexed under two keywords, the individual program name and the generic word print. Thus information on our program, PT, is arranged as follows:

```
" "  
"PT  
"SS  
<first line of text describing program PT>  
<second line of text describing program PT>  
.  
.  
.  
"PRINT  
<next line of text describing program PT>  
.  
.  
.  
<last line of text describing program PT.  
"XX
```

When a user enters

CI> GETKY, PT

at his terminal, the text in the file HELP.IDX that describes PT is flashed on his screen.

If, on the other hand, the user enters

CI> GETKY, PRINT

information on all four print programs will be displayed, one set at a time, with a prompt after each program's description giving the user the option of aborting the GETKY program or continuing with the display of the information on the next program.

To make the display on the screen easily readable during interactive use by not showing more than one screenful of data at a time, the SETKY command, "&, is inserted into the text every 15 or so lines. This command causes GETKY to pause the display at each "& and give a prompt allowing the user to display more text or abort the display.

A list of keywords under which help information is indexed in HELP.IDX will be displayed when a user enters

CI> GETKY, ? or

CI> GETKY, HELP

In fact, any time GETKY is run with a character string that is not a valid keyword in HELP.IDX, a list of valid keywords which bracket, in alphabetical (ASCII) order, the requested word will be displayed. Then the user may chose one of the keywords displayed, move forward or backward through the list, or halt GETKY.

Of course, GETKY allows the user the option of sending the text found under the keywords to a printer or to a file, as well as defaulting to the screen. But for this situation where a manual is also available, the screen is the main output device.

Because the intention was to generate a user's manual as well as to provide online help from this data file, a certain amount of forethought went into the ordering of the data file. First an introduction to the manual was written and placed at the beginning of the data file. Then help topics were added in the order in which they were to appear in the book. Finally SETKY and TEXED commands were inserted into the file. The same data file can be used for input to SETKY-GETKY and to TEXED

because all the commands for SETKY-GETKY begin with a double quote in column 1, and TEXED treats as comments records beginning with a double quote in column 1. On the other hand, all TEXED commands begin with a period in column 1, and SETKY-GETKY ignores all records beginning with a period in column 1 with two exceptions, .INDEX and .PAGE, which are interpreted in an analogous manner by both programs.

Thus the standard TEXED commands to define margins, page lengths, headings, chapter titles, etc., could be inserted in the HELP.IDX file without interfering with its keyed access via SETKY-GETKY. For example, all the topics relating to data communications were arranged together in the file preceded by a TEXED chapter heading for data communications. Then the entire HELP.IDX file was processed by the TEXED program to produce a manual complete with an introduction and a table of contents.

It is also possible to provide an index to the user's manual by using the .INDEX command. This command is acted upon by both TEXED and SETKY. In SETKY, the .INDEX command is an alternative method of referencing data by additional keys. The leading key, preceded by the pair of double quotes, is always required. However, if the user wants to have additional keys reference the data, they may be embedded within the body of the data, i.e., between the "SS and "XX records. These embedded keywords must be preceded with either a double quote, as was done with the keyword PRINT in the previous example, or .INDEX starting in column 1. When the data file is intended to be processed by TEXED, this option gives the creator of the file the ability to determine whether or not a keyword will go into the TEXED index. To ensure that the keyword will go into the index, use the .INDEX command. To have a keyword that is recognized by SETKY, but does not appear in the TEXED index, use the SETKY quote method of marking keywords.

Another useful feature of the SETKY-GETKY system is that it only accesses data that is preceded by a keyword and surrounded by "SS and "XX. Thus it is possible to include sections of text in the data file that are intended for the manual alone, such as the introduction, that do not appear in the online system.

Conversely it is possible to have online information available that is not to be included in the manual. For example, it is desirable to have many of the Hewlett-Packard supplied help files on commands available in the online system, but it is redundant to include them in our user's manual for our HP1000, since they are already described in the HP User's Manual. This is accomplished quite easily by taking advantage of the GETKY facility for executing run commands embedded in the data file.

For instance, to have information on the CI command, WD, available under one of the keywords in HELP.IDX, it is sufficient to include in the HELP.IDX file the following three SETKY commands:

```
" "  
"WD  
"RU, LI, /HELP/WD
```

Since these three records all begin with ", they will be ignored as comments by TEXED and not go into the manual. But when the user enters

```
CI> GETKY, WD
```

the CI program, LI, is scheduled by the embedded run command ("RU) and lists the contents of the HP-supplied help file, /HELP/WD.

The option also exists to make available as online information text that is stored in a different indexed data file on the computer. For example, it might be desirable to have online help available on using PC's without including it in the HP1000 user's manual. There could be another data file called PC.IDX, which contains information relating to PC's. Within this PC.IDX file, there could be a section of text that describes how to use the Vectra and is indexed under the keyword VECTRA. Then in HELP.IDX only the following three SETKY commands would be required:

```
" "  
"VECTRA  
"TR, PC.KEY
```

Then the program GETKY will automatically transfer to the PC.KEY key file to find the location of the data stored under the keyword VECTRA in the corresponding PC.IDX data file and then display the data indexed there.

The option of changing the keyword between data files also exists. For instance the same data indexed under VECTRA in PC.IDX could be displayed when the user enters

```
CI> GETKY, MICROCOMPUTER
```

if the following three SETKY COMMANDS were included in HELP.IDX

""

"MICROCOMPUTER

"TR, PC.KEY, VECTRA

In the case where a keyword references a relatively long section of text, it is desirable to be able to break up the text in two different ways: into relatively short sections, about 15 lines long, for the screen and into longer sections for pages of the manual. This can be accomplished by judiciously interspersing the "& and .PAGE commands so that a .PAGE command occurs only after two or three "& commands have been given. GETKY treats both commands the same way so it will pause the display at each occurrence of either command. TEXED, on the other hand, will ignore the "& commands and go to a new page only at the .PAGE command.

The information available both in the online system and in the user's manual can be easily changed or updated in both places by making the appropriate changes in the one data file, HELP.IDX. However, any time the data is modified, SETKY must be rerun to generate a new key file to be used by GETKY to locate the data. And, of course, TEXED should be rerun to incorporate the changes into the manual.

A corollary feature of this help system is the ability to generate online help when a user tries to run a program using an incorrect runstring. Assuming that an explanation of the program's correct usage is included in the data file, HELP.IDX, this explanation can easily be accessed from the program being run if each program is written to provide a special error routine. This error routine can call the subroutine, RuGetKy, from the SETKY-GETKY subroutine library to run GETKY with the program's own name as the key so that the information on its usage will be displayed from HELP.IDX. This is much easier than having to repeat an explanation of the program's usage within its own code. Also, when the directions for running a program are changed, the change only has to be documented in one place in HELP.IDX and the new directions will automatically be available in the general online help system, in the latest version of the user's manual, and from the program itself.

ACCESSING TABULAR DATA FROM AN APPLICATION PROGRAM

Another situation in which SETKY-GETKY can be very useful is in the retrieval of tabular data. It was used in our data center to access tables of chemical thermodynamic functions that were stored in a large data file, containing over 9000 records, on the HP1000. This file had been created before SETKY-GETKY was written. There were about 180

different tables of data in the file, called INTCOD. Each table was organized according to the same specified format. For example, table number 3 for O₂(g) was set up as follows:

```
ST:
T#:3
*
          CODATA Thermodynamic Tables
SP:  O2(g)
RE:  IVTAN DEC 83 FOR CODATA CTT (AUXILIARY DATA)
GS:  8.31448
CN:  0.1 MPa
*      Table 6.1.xx.  THERMODYNAMIC FUNCTIONS at 0.1 MPa: O2(g)
CH:    T,          ,   Cp   , -(G-H(0))/T,   S   ,   H-H(0)
UN:    K,          , J/(K.MOL), J/(K.MOL) , J/(K.MOL),   KJ/MOL
DT:  6

      0          0.000      0.000      0.000      0.000
     100        29.112    144.298    173.308    2.901
     200        29.128    164.424    193.489    5.813
     300        29.387    176.217    205.333    8.735
     400        30.108    184.606    213.871    11.706
     500        31.093    191.163    220.693    14.765
     600        32.093    196.576    226.451    17.925
          .
          .
          .
     3900        41.546    258.412    295.211    143.517
     4000        41.695    259.345    296.265    147.681
     298.15      29.378    176.038    205.152    8.680

$
*
          Uncertainties in Functions
ER:  5

      300          0.003      0.005      0.005      0.002
     1000          0.005      0.010      0.010      0.010
     4000          0.010      0.010      0.020      0.020

$
CC:  Functions from CODATA Key Values for Thermodynamics [84COD]
*    ΔfH0(0)/kJ.mol-1      = 0.0
*    ΔfH0(298.15 K)/kJ.mol-1 = 0.0
*    ΔfG0(298.15 K)/kJ.mol-1 = 0.0
CP: *Values at 100, 200, changed to match IVTAN Mar 11 '84 printout
SF: /GARVIN/CODATA/NAUXCA,O2(G)
ED:
```

To prepare these tables for publication, a processing program, RTOS, was written to retrieve the requested tables (each identified by a unique

table number), re-arrange their data into the required output format, and write the reformatted tables into a file for printing on a laser printer. Tables were processed in the order in which they were to be published, not the order in which they were stored in the original file, INTCOD.

Running the RTOS program was very time-consuming, taking about 40 to 45 minutes for 65 tables. The majority of time was taken up, not by the actual processing of the tables, but by the sequential search in INTCOD for the tables.

To speed up processing, we converted RTOS to use the programmatic interface available with SETKY-GETKY to retrieve the requested tables. First, SETKY commands were inserted in INTCOD so that the number of each table became a unique key to the table. Minor editing was required to insert the necessary SETKY commands into INTCOD and produce an indexed file, called INTCOD.IDX. Each table was preceded by "" and "<table no.> records. The ST: and ED: records in INTCOD were replaced by "SS and "XX, respectively. The T#: records were removed. The remainder of the table remained unchanged as shown below:

```
""
"003
"SS
*          CODATA Thermodynamic Tables
SP: O2(g)
RE: IVTAN DEC 83 FOR CODATA CTT (AUXILIARY DATA)
GS: 8.31448
CN: 0.1 MPa
*      Table 6.1.xx. THERMODYNAMIC FUNCTIONS at 0.1 MPa: O2(g)
CH:      T,      ,      Cp      , -(G-H(0))/T,      S      ,      H-H(0)
UN:      K,      , J/(K.MOL), J/(K.MOL) , J/(K.MOL),      KJ/MOL
DT: 6

      0          0.000      0.000      0.000      0.000
      100        29.112      144.298      173.308      2.901
      200        29.128      164.424      193.489      5.813
      300        29.387      176.217      205.333      8.735
      400        30.108      184.606      213.871      11.706
      500        31.093      191.163      220.693      14.765
      600        32.093      196.576      226.451      17.925

      .
      .
      .
      3900        41.546      258.412      295.211      143.517
      4000        41.695      259.345      296.265      147.681
```

	298.15	29.378	176.038	205.152	8.680
\$					
*	Uncertainties in Functions				
ER: 5					
	300	0.003	0.005	0.005	0.002
	1000	0.005	0.010	0.010	0.010
	4000	0.010	0.010	0.020	0.020
\$					
CC:	Functions from CODATA Key Values for Thermodynamics [84COD]				
*	$\Delta_f H^0(0)/\text{kJ.mol}^{-1} = 0.0$				
*	$\Delta_f H^0(298.15 \text{ K})/\text{kJ.mol}^{-1} = 0.0$				
*	$\Delta_f G^0(298.15 \text{ K})/\text{kJ.mol}^{-1} = 0.0$				
GP:	*Values at 100, 200, changed to match IVTAN Mar 11 '84 printout				
SF:	/GARVIN/CODATA/NAUXCA,O2(G)				
	"XX				

Then SETKY was run as follows:

```
CI> SETKY, INTCOD.IDX
```

to generate the key file, INTCOD.KEY.

The table processing program, RTOS, was modified so that the requested tables were retrieved from INTCOD.IDX by using the programmatic interface available with SETKY-GETKY, rather than by sequential reads of INTCOD. This modified version, CTABS, of RTOS uses the subroutine, OpenKy, to open both the data file, INTCOD.IDX, and the key file, INTCOD.KEY, with the following call:

```
call OpenKy (datafile, error)
```

where datafile is a character string set equal to INTCOD.IDX and error is an integer used to return an error value.

To position the data file at the location of the next table to be processed, the SETKY subroutine, PositionKy, was used:

```
call PositionKy (key, datafile, mode, error)
```

where key is the character string defining the keyword for which datafile is to be positioned. In this case, key is set equal to the table number desired. Datafile and error are as defined for OpenKy. Mode is an integer that specifies whether or not the key file should be rewound before the search is made for the key. In this case, it does not matter, as the table numbers are unique keys. However, when

PositionKy is to be used in a situation where identical keys reference different sets of data, mode is significant. If the key file is always rewound, PositionKy will always position the data file at the location of the data indexed by the first occurrence of the key. On the other hand, if the key file is not rewound, successive calls to PositionKy with the same key will locate successive sets of data referenced by that key.

Once the data file has been positioned at the start of the data for the next table to be processed, the ReadKy subroutine is called repeatedly to retrieve the data from the data file one record at a time:

```
call ReadKy (datafile, cbuf, length, mode, error)
```

Again datafile, in this case, is a character string set equal to INTCOD.IDX. Cbuf is a character string buffer in which the data retrieved from the data file by ReadKy is returned. Length is an integer parameter defining the length in bytes of cbuf. Mode determines the type of read to be performed. In CTABS, mode is set so that only data is returned. In word processing types of applications, mode could be set so that both data and TEXED commands, if there are any in the data file, are returned. Error is an error value returned by ReadKy to define the results of the read, such as, whether data is being returned or whether an end-of-data SETKY command was encountered.

After all the requested tables have been processed by CTABS, the data file, INTCOD.IDX, and the key file, INTCOD.KEY, are closed by a single call to CloseKy:

```
call CloseKy (datafile, error)
```

where datafile is set equal to INTCOD.IDX.

The completed program, CTABS, was run a few times on the data file, INTCOD.IDX, using the same list of 65 requested tables as were run from RTOS using the original data file, INTCOD. CTABS took only 8 to 9 minutes to run, compared with the 40 to 45 minutes it took RTOS to process the same tables. So a substantial savings of computer time resulted from the conversion.

In addition, once the data file is indexed on table numbers, it is easy to add a second key to each table, using the .INDEX command followed by the substance name for that table. For example, table number 3, which was already indexed by the key, 003, was also indexed by the key, 02(G), simply by inserting a record, .INDEX 02(G), following the SP: 02(G)

record in INTCOD.IDX. This doubled the number of keys to INTCOD.IDX. SETKY was rerun to generate an updated version of the key file, INTCOD.KEY. Then if a chemist is interested in viewing the table contained in INTCOD.IDX for a particular substance, for example, H2O(G), he only has to enter

```
CI> GETKY, H2O(G), INTCOD.KEY
```

to have it displayed.

For the cases where there is more than one table for the same substance, the user is given the opportunity to see each table in succession. For instance there are two tables for C(GRAPHITE) in INTCOD.IDX. To see both, enter

```
CI> GETKY, C(GRAPHITE), INTCOD.IDX
```

To obtain a listing of the same two tables on the printer, lu =6, enter

```
CI> GETKY, C(GRAPHITE), INTCOD.IDX, 6
```

SUMMARY

The SETKY and GETKY programs, along with the programmatic interface, provide a fairly simple, yet flexible system for indexing and accessing free formatted textual or tabular data on the computer. This system can be helpful to users at various levels.

- (1) GETKY can be run by users possessing minimal computer skills to retrieve keyed data, display it, print it, or capture it in a file.
- (2) SETKY can be run by users experienced in editing to set up keyed access to their own data files by inserting the appropriate SETKY commands.
- (3) The programmatic interface available with SETKY-GETKY can be used by programmers to access keyed data from within their own programs.

REFERENCES

- (1) "SETKY-GETKY Keyed Access System" is scheduled for inclusion in the CSL/1000 release for 1986 (release 2625) from INTEREX, 680 Almanor Avenue, Sunnyvale, CA 94086, and is available from the authors upon request.
- (2) Atre, S., *Data Base: Structured Techniques for Design, Performance, and Management*. John Wiley & Sons, Inc., New York, 1980. Page 197.
- (3) Bickham, D. and Neumann, D., "SETKY-GETKY - A keyed access system for the HP1000 - A user's guide," National Bureau of Standards, internal report available from the authors upon request.
- (4) GENIX, HELP, CMD, and RTE-6 are all Hewlett-Packard products and are referred to in the manual *RTE-6/VM Terminal User's Reference Manual* (December 1981), Part No. 92084-90004, Hewlett-Packard Co., 11000 Wolfe Road, Cupertino, CA 95014.
- (5) GRAFIT/1000 is a software product of Graphicus, Graphic User Systems, Inc., 160 Saratoga Avenue, Suite 32, Santa Clara, CA 95051.
- (6) "TEXED User's Guide, A Document Processor for the HP1000," documentation edited by Bill Hassell, adapted by Bruce H. Stowell, Jim Bridges, Bill Hassell, and John Johnson. The TEXED program and manual are available on the swap tape from the INTEREX Conference held in Washington, D.C., in September, 1985.
- (7) EDIT/1000 is a Hewlett-Packard product and is described in the manual *EDIT/1000 User's Guide* (August 1980), Part No. 92074-90001, Hewlett-Packard Co., 11000 Wolfe Road, Cupertino, CA 95014.

HOW DO THE USERS USE YOUR SYSTEM?

Donald A. Wright
Interactive Computer Technology
2069 Lake Elmo Avenue North
Lake Elmo, MN 55042 USA

INTRODUCTION

The HP/1000 accounting system maintains cumulative totals of logon time and CPU utilization for each user. But for the purposes of justifying a new system or the existence of a current one, or for analyzing how users' needs might be better served, more information is needed. This paper describes programs and a data base for collecting usage information and reporting that in a useful way. The following is collected for each session:

- * Logon/logoff times, CPU usage per session.
- * Time spent in each program and CPU usage per program.

The following may be reported for any time period:

- * Each user's programs with their CPU and usage times.
- * Each program's users with their CPU and usage times.
- * Overall summaries.

The system creates very little overhead of its own. The Image data base may also be examined by QUERY or any other inquiry program for ad-hoc reports. The package has been contributed to the Detroit Conference Swap Tape.

MANAGEMENT ISSUES

Technical computer systems are very often found in areas where computers are not the main business -- laboratories and manufacturing plants are typical locations. Often the people with overall management responsibility in such areas see the HP/1000 as a tool which aids in meeting their overall objectives. They want to know how the tool is being used and by whom. Without this information they find it

difficult to justify additional or even continuing costs. Managers are faced with these issues, among others:

- * Is a dedicated system-manager really justified?
- * Shall we purchase (or renew) software maintenance agreements?
- * How about hardware maintenance?
- * Are the requests for more peripherals (terminals, disks, plotters, printers, etc.) justified?
- * Do we really need another new system?

These issues can be much more easily resolved if two questions can be answered in an understandable way:

- * Who, What, How much?
- * What, Who, How much?

The first question really asks "Who is using the computer, what programs are being used by those users, and how much are they used by those users?". The second question asks for exactly the same information, but sorted first by program and then by user.

The RTE accounting system provides only Who and How Much in total. It simply doesn't save enough information for any purpose except for assigning costs.

PROPOSED SOLUTION

For this solution it was decided to keep the following information in an Image data base for each user session:

- * Start and length of the session.
- * For each program the user runs:
 - * Time of first and last runs.
 - * CPU utilization.
 - * Total usage time.

It was decided that it was NOT important to know how many different times a user ran a program - but the total amount of time that a user uses a program may be quite interesting.

The initial need for this solution was perceived by managers of a very large F-Series system running RTE-6. The solution described can, in principal, be applied to RTE-A and may have been by conference time.

Data Collection:

Requirements for the data-collection process were defined as follows:

- * The process must start automatically at boot, e.g. from the WELCOM file.
- * The data collection must be a non-intrusive measurement. In other words, it must not substantially affect the measurement by its own operation and existence. Therefore the data-collection program must not hog or even be a dominant user of any resource - CPU, disk, or Image subsystem.
- * Every reasonable means must be used to accurately identify each user and the true name of each program.
- * We decided not to try to rewrite RTE to collect the information, but rather to use programs operating outside the operating system.
- * A sampling technique would suffice if the sampling interval could be adjusted and could be short enough.
- * Non-session programs can be ignored. On the system for which this monitor program was designed, very few users run programs in batch or detached, and those few are genuinely special cases.
- * Group information was not of interest on this particular system, where groups are not really used at all.

Data Presentation:

QUERY is used for ad-hoc reports where the data must be selected in a way not otherwise provided for. But the bread-and-butter report program for management's use is designed to meet these requirements:

- * Reporting period must be selectable, from one day up.

- * The Who, What, How much report:

For each user the programs are tabulated in alphabetical order, with usage information:

- * Program name
- * CPU seconds for the program.
- * Usage seconds for the program.

Total CPU seconds and connect time are also reported.

- * The What, Who, How much report:

For each program its users are tabulated in alphabetical order, with usage information:

- * Username.
- * CPU seconds for this user.
- * Usage seconds for this user.

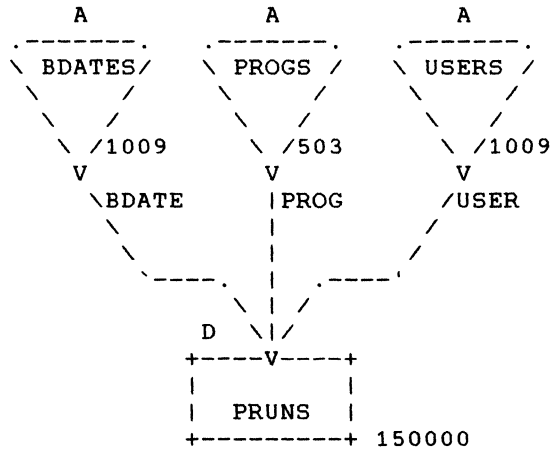
Total CPU seconds and usage time for the program are also reported.

- * Total number of different programs and users for the reporting period.

- * The data-base search, sort, and reporting must be fast and easy enough that managers will actually use it.

DATA BASE STRUCTURE

This rather simple data-base structure was selected:



BDATE	X6	(k)	Beginning date, YYMMDD
BTIME	X6		Beginning time, hhmmss
EDATE	X6		Ending date, YYMMDD
ETIME	X6		Ending time, hhmmss
INTERV	I1		Interval between observations, seconds
PNAME	X6		ID-segment (runtime) program name
PROG	X6	(k)	Actual program name, before cloning
SECCPU	R2		Seconds of CPU usage, this program/session
SECON	R2		Seconds first observation to last
SECUSE	R2		Seconds program-usage/session-connect time
SESNUM	I1		Session number
USER	X10	(k)	User's logon name

This structure contains three automatic master data sets linked to a single detail data set. This provides for very fast searching on any of the three key items Date, Program, or User. In practice it eliminates the need for serial searches in the large majority of data extractions.

The detail data set PRUNS contains one entry for each user session, and one for each program that the user ran during that session. The session entry is distinguished from program entries by the value SESSN in the PROG item. In this case, the user's runtime primary program name is inserted in the PNAME item.

Thus when a typical session ends, the data-collection program puts one session entry and several program entries in the data base.

Note that the current implementation uses Image 1, not Image 2. Presumably Image 2 will work as well.

DATA COLLECTION

The MONITOR program is scheduled from the WELCOM file. It initializes itself, places itself in the time list, and begins the first data-collection run. When finished, it terminates saving resources. Since it doesn't lock itself into memory, it is swappable between runs, although its priority would normally cause some other program to be swapped.

It reads all usernames from disk into local memory on the very first run and each hour thereafter, on the hour.

The standard data-collection interval is 10 seconds, but this can be adjusted by a runstring parameter to any number of seconds.

Each time MONITOR runs it goes through three phases:

- * Data collection.
- * Accounts resolution (comparing current sessions with previous ones).
- * Data logging (writing to the data base).

Data Collection:

The accounts system is first locked, so that no changes can occur while the sessions are being examined.

Every session-control block is examined. If it is new, an entry is built in the Local Session table. Then the current CPU utilization is recorded, the ID-segment address of the currently-executing program is saved, and the session is flagged as having been found.

The accounts system is unlocked at the end of this search, which typically requires only two or three milliseconds. No disk accesses are ever required, nor are any privileged operations.

Accounts Resolution:

In this phase, the accounts system is unlocked and disk accesses are permitted, though they are required only occasionally. Each entry in the Local Session Table is examined as follows:

- * Ignore the session if not found this run, as it will be dealt with in the data logging phase.
- * If this is a new session entry this run, build an initial entry and put it in the data base.
- * Find the program's actual name:
 - * Use the ID-segment name if it is a memory-resident program or if the ID-segment indicates that the name is not a clone name.
 - * Else look in the table of already-identified names for a program with exactly the same disk address.
 - * If not there and the disk address indicates that the program is not in FMP file space, search for an identical permanent ID-segment and use that name.
 - * Last resort - a disk operation. Go to the actual type-6 file on disk and pull the name out of that file. Put the resulting name in the local table of already-identified names.
- * Search the session's linked-list of program names for this name, add if not found.
- * Accumulate the statistics for the session and program. The CPU utilization and time interval for the time between the past and current run are allocated to the currently-executing program. Note that this can create an error proportional to the sampling interval - a granularity error.

Data Logging:

This involves another scan of the Local Session Table, looking for sessions which have disappeared from the system's session tables and thus were not found on this run. When such an entry is detected, or if the program is shutting down (described later), the following steps are taken:

- * The data base is locked.
- * The initial session entry, created when the session was first detected, is recovered with a DBGET.
- * The final session statistics are integrated into this entry and it is DBUPdated.
- * For each program in the session's linked list, a program entry is built in memory and DBPUT in the data base.
- * The data base is unlocked, and the Local Session Table entry is deallocated for re-use.

Other MONITOR Features:

The runstring for the MONITOR program is:

```
RU MONITOR [interval [password ]]
```

Where interval is the number of seconds between MONITOR runs or the characters EN or SD. EN or SD causes MONITOR to shut itself down, closing out all open Local Session Table entries as if they actually logged off. This is useful during debugging, and if the system manager intends to boot the system. If SD is specified, the RTE-6 accounts system will actually be shut down, preventing further logons. When the system is booted, the accounts system will be available for logon again.

If EN or SD is specified, the password must be supplied.

Note: If MONITOR encounters any kind of fatal error during operation, such as an Image or EXEC error, it will enter the shutdown mode and attempt to close out all sessions before terminating.

It runs at a priority of 29, which gives it priority over most DS monitors but not over the monitors which are time-critical. If it is not allowed to run at its scheduled time no real harm is done - when it does run the actual interval rather than the scheduled interval will be used in incrementing usage time counts.

DATA PRESENTATION

The data-presentation program USAGE is interactive, and has three operational phases:

- * Operator input.
- * Data base search.
- * Formatted output, to file or device.

Operator Input:

Prompts are issued for the following information:

- * Beginning search date.
- * Ending search date.
- * Output file or device for the report by User. If none is supplied, no report by User will be issued.
- * Output file or device for the report by Program. If none is supplied, no such report will be issued.
- * Output report format: Full or Summary.

Data Base Search:

The data base is always searched on the BDATE (beginning date) chains, starting with the beginning search date and incrementing through the ending search date. Thus every entry retrieved is a needed entry. For each entry found, the following steps are executed:

- * Program-name translation is done. This is a process which allows program names known to be incorrect in the data base to be translated to proper ones. The translation file is created and updated with EDIT, so that this represents an after-the-fact way to correct reports on a cut-and-try basis if necessary.
- * Program and User names are inserted into sorted tables with no duplicates.

* For each user, a linked-list is maintained in EMA of all associated programs and their statistics.

* Likewise for each program, a linked-list is maintained in EMA of all associated users and their statistics.

The speed of this process is constrained by the data-base disk accesses. The name-sorting and list-maintenance components appear to take almost no time by comparison. Binary searches are used on sorted tables, serial searches on the linked lists.

Report Output:

Production of the reports is extremely straightforward. The sorted tables of Users and Programs are written in order, with the linked-lists of associated programs and users sorted in a simple manner and written along with them. Reports are written at a speed dependent upon the output device, whether disk or something else.

The following is a portion of an actual report:

HP/1000 USAGE 860603 through 860609

USERNAME	PROGRAM	CPU SEC	%	ELAPSED SEC	%
AEJ	CI	0.00	0.0	10.00	.0
	CONFR	6.60	.1	26440.00	17.6
	D.RTR	2.55	.0	30.00	.0
	EMUL8	4.21	.1	330.00	.2
	METER	6652.92	99.7	123200.00	82.1
	TYPE	4.83	.1	50.00	.0
		6671.11	47.0	150060.00	25.5
DJL	BASIC	3.14	.5	70.00	.4
	CI	2.17	.3	250.00	1.3
	CONFR	13.93	2.0	150.00	.8
	D.RTR	77.69	11.2	420.00	2.1
	DL	7.49	1.1	60.00	.3
	GRAFI	42.19	6.1	430.00	2.2
	GRMSG	20.39	2.9	80.00	.4
	SKETC	524.23	75.8	18350.00	92.6
		691.23	4.9	19810.00	3.4

RNR	ACCTS	.95	.1	20.00	.0
	CALC	3.50	.2	460.00	.2
	CI	18.05	1.2	1330.00	.7
	CIX	3.60	.2	50.00	.0
	CONFR	109.16	7.3	41386.00	22.4
	D.RTR	118.16	7.9	580.00	.3
	DISCV	.59	.0	20.00	.0
	DL	17.14	1.1	242.00	.1
	EDIT	7.65	.5	2050.00	1.1
	EMUL8	42.26	2.8	12420.00	6.7
	FOWN	111.79	7.4	310.00	.2
	IOMAP	.32	.0	60.00	.0
	KERMI	8.38	.6	1140.00	.6
	LI	19.25	1.3	870.00	.5
	METER	1031.56	68.7	122872.00	66.5
	TF	2.27	.2	900.00	.5
	TYPE	4.88	.3	40.00	.0
	USAGE	2.35	.2	100.00	.1
		-----	-----	-----	-----
		1501.86	10.6	184850.00	31.4

Grand totals:	14207.18	589235.00
Programs: 32		
Users: 29		

ACTUAL USE

In use, it appears that the MONITOR program actually does achieve the objective of non-intrusive measurement. Timing tests showed it to require 20 to 30 milliseconds of elapsed time for most runs, with a normal logon load of 5 to 10 users. Occasionally this number increases significantly when data-base accesses are necessary. Thus when the interval between runs is 10 seconds, MONITOR requires only about 0.3% of the CPU's time and presumably an even smaller fraction of the disk accesses. Its priority can be adjusted to avoid harming other real-time processes if necessary, without substantially impairing its accuracy. These tests were done on a very large F-series system running RTE-6 C.83.

The reporting program USAGE achieves its objectives with the maximum possible speed, limited only by the disk accesses required to read the data from the data base.

Using C For Portable Programming

Tim Chase

Corporate Computer Systems, Inc.
33 West Main Street
Holmdel, New Jersey 07733
U.S.A.

(201)946-3800

642672 CCSHOLM

Increasing software development costs, changing computer architectures and greater user reliance on standardized application packages are forcing close examination of software portability. Many feel that the C programming language is an excellent tool for developing applications which are to be ported from one computer to another. Unfortunately, although the literature abounds with introductions to programming in C, there are few references which provide insight as to how applications developed in C may be structured so as to increase their portability.

C is currently a widely used programming language and with standardization efforts on the part of ANSI, we can be assured in the near future that C will become a "pure" language on all computers which offer conforming implementations. Because of C's features, programs written in C can be easily moved between computers which support the language. As with all current programming languages, C, is not automatically portable. Applications developed in C require conscious use of C's features in order to insure portability. Although C makes writing portable programs easier, the task is still not "for free." The programmer must know what he is doing in order to achieve true portability.

What is true portability?

The definition of portability varies from person to person. We will define a portable program to be any program which may be moved from one computer to another with less effort than would be necessary to develop it again from scratch for the second machine.

This definition often surprises some people who believe that a portable program is one which runs on any computer immediately after it is recompiled. In actuality, portability covers a range of programs. At one end of the range lie programs which, in fact, do run after a recompile. At the other end are programs which must be substantially rewritten in order to port them. When dealing with software portability, it is important to correctly set your expectation level. Someone who advertises an "easily ported" program may be over-selling capabilities because there is no accepted definition of portability.

How is portability achieved?

In essence, a program is made portable by:

1. Using a standard language.

This means that the application must be implemented in a language which is available on all of the machines the program is to be ported to. It will do you no good to write an application in your favorite language if that language is not implemented on a potential target machine. Fortunately for the C developer, the C language is offered on virtually all modern computers.

2. Isolating machine dependencies.

The well written portable program is broken into two parts. The first part is the "machine independent" part while the second part is the "machine dependent" part. The plan is to rewrite the machine dependent part each time the code is ported and to not rewrite the machine independent part. To make the port job attractive, the machine dependent part should be (much) smaller than the machine independent part.

In this two step approach the first step (selection of a language) is simple. For the purposes of this paper the language will be C. The reason for this choice for an increasing number of developers is because C makes the work involved in the second step simpler. This is because C offers a number of unique features which make the isolation of machine dependencies easier than in other languages. We will show these features at work later in this paper and we will draw from experience in porting programs between HP/3000's and HP/1000's as well as non-HP computers.

How are machine dependencies found?

That's the big question. The comic Steve Martin has a routine in which he claims to be able to tell you how to get a million dollars and not pay any taxes on it. The first step is "get a million dollars." Mr. Martin offers no more direction than that as phase one of his tax avoidance scheme. Our advice on finding machine dependencies will be more long-winded than Mr. Martin's punch line, but after all the words settle, the answer will be about the same -- find them.

Locating the machine dependencies is the art behind writing portable programs. There are guide lines. We can tell you where to look out for machine dependencies, but there are no cut and dried rules. Although this paper concludes with some general rules, experience will be your best teacher. The more computers you port your software to, the better you will become at writing portable programs. The act of porting software actually makes it more portable. Having developed several C compilers for HP equipment, we have been approached by a number of software suppliers who want to port their applications. Those who have ported their software to a number of other computers usually have no problems while those who have never ported their software are often con-

fronted by profound problems.

Using C does not insure that an application will port. Further, just because something is written in C does not mean that it will port easily. The reality is that portability is something which comes from extra care, extra development dollars and extra creativity during design and programming. Using C, however, is a good way to minimize these extras.

If you are beginning to think that developing a portable program is more expensive than developing a non-portable program, you are right. Clearly there are times when it does not matter at all if a program is portable. Still, many of the techniques that are used in developing portable programs have application in non-portable software development. They tend to make programs more easily maintainable and more easily understood.

Where to look for machine dependencies.

Although it is impossible to present a complete list, some of the places which usually cause portability problems include:

Word size and alignment

If your program assumes that memory locations have a given number of bits in them, it will not port to computers which have different sized words. This is one of the most important things to remember when writing programs for the 1000/3000 computers which must port to the Spectrum machines.

Storage alignment is the requirement placed on some computer designs that storage units begin on certain addressing boundaries. For example, if a computer has a byte oriented memory, a 16 or 32 bit word may only be allowed to begin on a special "boundary" or multiple of bytes. A 16 bit word might only be allowed to start on an even byte address. This is a requirement of the HP/3000 and the HP/1000 although because both computers are word addressed, programmers seldom think of this as an alignment rule.

Alignment rules become important when you are designing "data aggregates" or collections of objects which are to be treated as a whole. C's struct feature allows users to define records of data objects. If your program makes assumptions about how those objects are physically placed in memory, chances are it won't port.

Character usage.

Character sets vary from one computer to another. Many computers are ASCII based, but others may use different coding schemes. For example, IBM is generally EBCDIC. This means that algorithms which rely on a given character being represented in a given way, will not port. For example, if you want to be able to retrieve a value from an array X by indexing with a character 'A':

```
value = X['A'];
```

The initial value of the X array will depend on the character set of the computer you are porting to. Such a program written for the HP/3000 would not port to most IBM machines.

Another problem with characters is how many can be placed into a single word of computer memory. A 16 bit computer would allow 2 while a 32 bit computer would allow 4. Again, programs making use of these facts will not port.

Arithmetic portability.

If you are writing numerical applications, the values obtained on one computer could differ substantially from those generated by the identical program on another computer. This is because of the precision offered by different machines. For example, a C integer (type int) offers 16 bits of precision on the HP/1000, but 32 bits of precision on the Spectrum processors.

Arithmetic portability problems, strangely enough, are found even in more "arithmetic based" languages like PASCAL and ADA. This is because range checking is usually performed at value assignment time rather than after each intermediate expression. This means that intermediate expressions could produce implementation dependent results which would not be caught by runtime checking. The result would be a program which could produce different results on different processors -- clearly not portable.

Operating system environment

This is perhaps one of the single largest factors effecting portability. If the operating system services which your application uses are not found on a potential target machine, the port job will be difficult. One common problem with non-portable C applications is the assumption by the developer that "where there's C there's UNIX."

Getting back to basics, UNIX is an operating system and C is a language. Although C and UNIX are often found in the same places, the existence of C does not in any way imply the existence of UNIX. UNIX, as an operating system, offers many of the same features found in most operating systems, but it often offers them in slightly weird ways (Bell Labs, don'tcha know?) Applications which use these features are doomed to be difficult to port.

File and data base systems.

How files are accessed differs from system to system. Again, UNIX offers an ideal example. The file system found on UNIX is different from "normal" file systems because of its file naming and its file organization. Applications relying on a UNIX file system present porting challenges. Likewise, applications which rely on a certain data base system can only be ported to computer environments which support that system. If you call IMAGE in your application and you want to get a version working on a DEC VAX you are in for a fun evening.

So what's a programmer to do?

What we have said up to now is that programming a portable application is simple if you avoid characters and arithmetic, don't call the operating system, don't use files and stay away from data bases. The only problem is that if you develop by these rules, you won't be able to write a very interesting application. The real answer is not to avoid these areas, but rather to recognize them as problem spots and to design your application so that you can still use the features of the computer, yet do so in a machine independent way. This is done through a technique we call "abstract interfaces".

An abstract interface is a fancy name for finding out the essential requirements you need for your application and then writing code to interface those requirements to the resources offered by your target computer.

When you stop to think about it, this is what is being done for you each time you use a compiler for any language. If you are a FORTRAN programmer, you think of the computer in terms of FORTRAN. You really don't care how the computer does its work, you are programming an "abstract" FORTRAN machine. The compiler, in this case, is the abstract interface you are using so that you, as a programmer, are portable from computer to computer. Abstract interfaces which you develop as part of your portable application behave in the same way and address differences in computers which are not covered by the language you are using.

When you develop an abstract interface, you can call it from your application. The abstract interface will be rewritten for each computer you are going to port your application to. The abstract interface becomes the machine dependent code. Let's look at a concrete example of this.

Suppose you want to develop a C program which will execute on both the HP/1000 and the HP/3000. Assume, for the moment, that you do not want to use the standard I/O library which comes with the C compiler (This I/O library is, in fact, an abstract interface defined by the designers of C which provides standard access to the file system of the host computer). In designing your program, you discover that you will have to make use of disc files. Remembering every word of this paper, you realize that file usage is a place where you must look for machine dependencies. Sure enough, when you compare the routines used for file access on the RTE and the MPE you discover they are different -- very

different. How do you make an application cope with the differences?

First, you look at your application's needs. Both the RTE and MPE support a wide range of file operations, but will you need access to all of them? Chances are you won't. So you select the set of features which you really need. Remember, each feature you select contributes to the size of the abstract interface and will therefore make the port job more difficult.

For our example, we discover that we will need file open, file create, file close, file read and file write. Other applications might need other functions like rewind and purge, but our example application is simple so we only include the features we need.

Next, we design the abstract interface for these functions. The question is what will each file routine do. Again, look at the application and pick only the features necessary to do the job. More features mean more work which means harder porting.

For the sake of brevity, we will only look at the file open routine. If we break the open down to its essential functionality, its job is to map a file name into an internal file descriptor. The descriptor can then be passed to other file routines for reading and writing. The open call looks like this:

```
descriptor = my_open(name);
```

The descriptor, however, is quite machine dependent. On the HP/1000's RTE a descriptor is a 144 word data control buffer while on the HP/3000's MPE the descriptor is a single 16 bit word. C offers features enabling you to define new data types which can be different on different machines. This is done through the use of the "typedef". We might make the following typedef definitions. For the HP/1000:

```
typedef struct {int dcb[144];} file_desc;
```

and for the HP/3000:

```
typedef int file_desc;
```

This has defined a new data type in terms of existing data types. The name of the new type is "file_desc" and it behaves much the same as any other data type in the language. Once defined, it may be used. Those uses are independent of the definition. We could write code like the following:

```
#include "dependent.h"

extern file_desc my_open();

main()
{
    file_desc input;
    file_desc output;

    input = my_open(in_file_name);
    output = my_open(out_file_name);
    :
}
```

The basic technique here is to gather together all of the machine dependent definitions (the typedef for example) and to place them into a C "include" file. In this case, the include file is called "dependent.h". There will be one include file for each target machine. When the above program is compiled on the HP/1000, the compiler will read the HP/1000 version of the dependent.h include file. When compiled on the HP/3000, the compiler reads the dependent.h file for the 3000. The compiler changes the object code output as a result of the different definitions found in the dependent.h include file.

The job of porting becomes one of rewriting the dependent include files and rewriting the abstract interfaces.

What are C's portability related features?

This is a tough question because any feature in the language could ultimately be used to construct an abstract interface. There are, though, some features which are especially useful in writing portable programs. What follows is a listing of these features and a discussion of each with respect to their use in portable programming.

The #include

The include statement is part of C's macro pre-pass. It causes a file to be merged at some point in a source program during compilation. Its primary use in portable programming is to isolate machine dependent definitions from machine independent code. All machine dependent definitions are collected together in to one (or more) include file(s) which are written for each target implementation. Include files of this type usually contain the typedefs, #defines and structure definitions which are required to implement the necessary abstract interfaces.

The #if

This is another pre-processor feature. It is used to conditionally control compilation of source code. For example,

```
#if HP_1000
    exec(2, 1, mesg, -12);
#elif HP_3000
    print( (char*)mesg, -12, 0);
#endif
```

In this program fragment, if the pre-processor variable HP_1000 is defined, then the EXEC call will be compiled. If the HP_3000 variable is defined, then the PRINT intrinsic will be compiled. Using the conditional compilation feature for portability is obvious, but it is generally not a good idea. The resulting source code is difficult to read. Defining a macro would be a better way to achieve the same results as we shall see.

The #define.

The #define statement is part of the pre-pass which defines macros. Macros are an important way in which C programmers write portable programs which are efficient. The define statement may be used to declare two different types of macros: those with parameters and those without.

Macros without parameters are useful for separating machine dependent constants from the pure portable code. For example, a data base file name might be defined in a machine dependent include file by using a macro. On the 1000 it might look like:

```
#define DATA_BASE_NAME "/system/roots/data.dbs"
```

While on the 3000 it might look like:

```
#define DATA_BASE_NAME "DATA.PUB.DBASE"
```

If we wanted to write portable code to open the data base file as per our file system example we would write:

```
data_base = my_open(DATA_BASE_NAME);
```

The define processor in the pre-pass would take care of all of the machine dependencies.

The second form of macro is one which has parameters. This is more complex, but much more powerful. Again, the machine dependent macros can be defined in the machine dependent include files. Assume we want to make an abstract interface to tell the operator something. The routine will be called "tell". The tell function will accept a string as its argument. The string will be printed to the operator's terminal.

What we have just done is to describe the abstract interface for

printing a string to the operator. Now we must implement it. There are two approaches. The first approach is to actually write a subroutine which is rewritten each time the application is ported. This is what we did for the "my_open" routine. If the interface is simple enough we might be able to implement it as a macro. This will increase the efficiency of the interface because there will be no subroutine call. Again, include files are used to contain the dependent code:

HP/1000 include file

```
extern int exec();
#define tell(s) exec(2, 1, *(int*)s, -strlen(s))
```

HP/3000 include file

```
#pragma intrinsic print
#define tell(s) print(s, -strlen(s), 0)
```

The portable programmer may feel free to use the tell macro whenever a message is to be sent to the operator:

```
tell("Hello, how's it going?");
```

Not only is this usage portable across both the 1000 and the 3000, it is efficient and much easier to read than using the #if construct.

Most C compilers support some form of embedded assembly language. This is useful for getting at special instructions which perform certain functions. It would appear impossible to use embedded assembly language in a portable program. But using abstract interfaces and C features makes it easy.

Take, for example, a move byte operation. The HP/3000 has a move byte instruction. Most HP/1000 computers also have a move byte, but some don't. The abstract interface we will design is:

```
move_bytes(from, to, number);
```

"From" is the source character pointer, "to" is the destination character pointer and "number" is the number of bytes. We can define the following machine dependent include files:

For the HP/1000:

```
#if OLD_HP
extern do_move();
#define move_bytes(f, t, n) do_move(f, t, n);
#else
#define move_bytes(f, t, n) asm { \
    lda f; \
    ldb t; \
    ldx n; \
    mb00; }
#endif
```

Notice that the `#if` is used in the include file to check the pre-pass variable "OLD_HP". If the OLD_HP flag has been set, then the external routine "do_move" is called to do the move. If the flag is not set, then the macro `move_bytes` is defined to generate in-line code to access the `mb00` instruction.

The include file for the HP/3000 is:

```
#define move_bytes(f, t, n) \  
    _TOS_ = (unsigned) f; \  
    _TOS_ = (unsigned) t; \  
    _TOS_ = (unsigned) n; \  
    asm(0020063);
```

Again, the application now may be written which uses the "move_bytes" feature regardless of the computer it will ultimately run on. Because macros have been used to implement the abstract interface, the resulting code is very efficient.

Structures, unions, typedefs, and bit fields

These four powerful features may be put in a single class with regards to portability. They enable the user to remove the definition of data from the algorithm which works on the data. This is key to the development of a successful abstract interface. The definition of data is often the biggest difference between computers. The "what is being done" part is usually portable if we can develop a description of the data.

Some languages make the separation of data and algorithm difficult. How many FORTRAN programs have you seen where integers are multiplied by powers of 2 in order to shift bit fields to appropriate locations? This type of programming intermixes the definition of data and the definition of the algorithms to access that data in ways which are extremely difficult to find and change.

The typedef is a feature which provides for the separation of data definitions and program definitions. As we saw in the file open example, the typedef can be used to define machine dependent arguments to abstract interfaces. Another interesting use for the typedef is in controlling integer precision.

The C language supports names for types which have nothing to do with the precision of the types. For example, the type "int" on the HP/1000 means a 16 bit word while the same type on the Spectrum means a 32 bit word. This can cause problems if the programmer specifically wants a certain representation. It is difficult to count to 123,423 with only 16 bits.

Typedef's defining new types can be made part of the the machine dependent include files written for each target computer. An example on the HP/1000 and the HP/Spectrum might be:

HP/1000 Include file

Spectrum Include file

```
typedef int bits16;
typedef long bits32;
```

```
typedef short bits16;
typedef int bits32;
```

If we write the following in the portable portion of the program

```
bits32 a[20];
```

then we are specifying the allocation of an array of 20 elements named "a". Each element of the array will contain 32 bits of precision. On the HP/1000, the definition of the type "bits32" is a long. On the 1000, a long is implemented using double precision integer arithmetic. On the Spectrum (a 32 bit machine), the same type name has been defined to be an int. A Spectrum int is 32 bits.

If we write the declaration:

```
bits16 b[20];
```

We will allocate an array b which also has 20 elements, but each of b's elements will have 16 bit representation.

A good example of the use of C structures is in the solution of the "NUXI" (pronounced nux-ee) problem. The DEC PDP-11 numbers its bytes "backward" from the HP/1000 and HP/3000:

PDP/11	HP/1000 HP/3000
+-----+-----+	+-----+-----+
byte 1 byte 0	byte 0 byte 1
+-----+-----+	+-----+-----+

The problem is that ASCII text stored into computer words gets reversed on the PDP/11 (unless you are a DEC programmer, then it gets reversed on the HP computers!) If an integer "x" contains two ASCII characters and we mask off the high order 8 bits expecting to get the "first" byte, we will discover the "second" byte if the program runs on the PDP/11. For example, the C expression:

```
(x >> 8) & 0377
```

Returns byte 1 on the PDP/11 and byte 0 on the HPs. It is this odd property which gives the problem its name. If the string "UNIX" is put into an array and read out on the "wrong" machine it becomes "NUXI" -- (ahh, Bell Labs again).

Anyway, the NUXI problem provides a good example of separation of data and algorithm. The "abstract interface" into the data is this: first, we want to be able to get at a word as a 16 bit integer and second, we want to be able to find the "first" ASCII character in the word regardless of the machine organization.

Unsurprisingly (by now) this can be done with code in machine dependent include files:

HP/3000 Include file

```
union charloc {
    struct {
        char first;
        char second;
    } c;
    int i;
} x;
```

PDP/11 Include file

```
union charloc {
    struct {
        char second;
        char first;
    } c;
    int i;
} x;
```

The include file on the HP/3000 indicates that `x.c.first` is the most significant 8 bits of an integer, while the same expression on the PDP/11 indicates the least significant 8 bits. We may now write the following portable code:

```
if(x.c.first == 'A') ....
```

When compiled on the HP/3000, the if statement tests the upper byte of the integer, yet when compiled on the PDP/11 (using the PDP/11's machine dependent include file), the lower byte will be tested.

The union enables the object "x" to be viewed either as an integer or as a structure of 2 characters. Notice the important fact that the "." operator used to select the appropriate component of "x" knows how to access the data based on the definition of x. Contrast this with coding the access method in the program:

```
(x >> 8) & 0377;
```

Coding the access method into the program is clearly non-portable. All of the places in the program where the shift and mask technique is used must be located and changed each time the program is ported.

The need to use shifting and masking to extract and deposit strings of bits from words is completely eliminated by C's bit fields. A structure may be defined which has members which are actually bit fields within the same computer word. For example, the structure

```
struct {
    int i;
    int j:4;
    int k:6;
} x;
```

Defines x to be a collection of 3 data objects. The first is a single integer called "i" while the second and third objects are bit fields of 3 and 5 bits each:

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| j field  |   k field   |      unused   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

To set the "k" field of x to a value of 47 we would write

x.k = 47;

There is no source code which deals with the fact that k is a field within the object x. In fact, if for reasons of portability, we were to change the definition of the structure so that k were no longer a field, but rather a full integer, only the definition would change. We would not have to change source code. Give me some rules already!

We have gone through a definition of portability, we have described the use of the abstract interface and we have shown you some of the features of C which are especially useful for portable programming. Although impossible to make complete, we will finish with a set of "rules" which can be used as guidelines in developing a portable program. Some rules are quite definite while others are quite vague. Such is the current state of the portable art.

1. Design before you code.

Sounds like motherhood and the flag, but many portability problems can be designed out at the earliest stages of a project. Recognize the "problem areas" we pointed out and make sure that your design addresses them. Generally, the most troublesome problems stem from choices made early in the implementation cycle.

2. Know your target computers.

If you have a good idea what computers you are going to port your software to, you can make the wisest implementation decisions. This is not often possible, but if you know a good sub-set of target machines, it will point out what sorts of things your design will have to accommodate.

3. Use features wisely.

Just because a system you are programming on has some nice feature, does not mean you are obligated to use it. In fact the "nicer" a feature is probably the more obscure and non-portable it is. Watch out for special language "enhancements" which are supposed to make your implementation better. Are they generally available? Special features should be isolated with abstract interfaces so they can be programmed around if they are not available.

4. Recognize and use the lowest common denominator.

If your target set of computers includes one with a 64K address space and one with a virtual address space, you'd better design for the smaller machine. If one computer only supports fixed sized files and another supports fixed and variable sized files, you'd better design using only fixed sized files. Again, knowing where you're going helps here.

5. Use standard libraries.

ANSI is doing a lot to standardize the I/O libraries found in C. Some implementations brag that they have all these weird and wonderful functions which make programming easier -- they also make it non-portable.

6. Avoid system calls like the plague.

Do not make any direct system calls. Instead abstract them either through macros or by writing intermediate code. Make sure that you have found the really important features of the system call. If, for example, you have a open function which will open a file and return the time of day with one call, ask yourself if you really need the time of day. If you don't, don't put that part of the feature in your abstract interface to the open function. The simpler the better.

One of the really gray areas is in shared memory functions and process management. There are as many (different) implementations for these features as there are operating systems.

7. Develop style rules and stick with them.

Remember, portable programs are always being rewritten. They must be clear and easily understood. Adopting a programming style and sticking with it makes all code modules appear uniform. Programmers can quickly become accustomed to where things will be and how they will be written.

8. Separate dependencies into include files.

Isolate machine dependencies into different include files. For example, perhaps an include file named "dependent.h" could be used. This should contain all of your machine dependent definitions.

9. Avoid coding "data access" in your program.

Let the compiler do this work for you. Use the struct, union and typedef features to describe the data you are using. Often you can change these definitions and your programs will remain unchanged. If you must code shift/mask type operations, make them portable. For example, to clear the low order 3 bits of an integer you might write

```
x & 0177770;
```

This is not portable because it assumes that x is 16 bits. A better way to write this would be:

```
x & ~07;
```

The compiler will make the appropriate sized constant for you.

10. Document assumptions.

If you are assuming certain features are present, document those assumptions. One of the by-products of a portable program is the "port manual." This is a document which describes what has to be done to successfully port the program.

11. Write test programs to check out features.

If you expect a certain feature to be a certain way, write a simple test program to check the feature. Before porting you can run the test to get a feel for what the target compiler is doing.

12. Don't "hard code" constants.

You should never code constants directly into source programs. Instead, use the macro facility of C to give constants names.

13. Use short external names and don't mix upper and lower case.

Different C compilers (and system loaders) do different things with external names. For example, CCS/C on the 3000 allows external names up to 31 characters in length, but CCS/C on the 1000 limits these names to 16 characters. Over all, 6 character external names appears safe. Fortunately, this size is increasing.

Conclusions

Our own experience and the experience of others has shown that by following these rules and by programming in C it is possible to develop highly portable software. A case in point is the CCS/C compiler. This compiler was developed for the HP/3000 on the HP/1000. When completed, the 1000 version of the compiler compiled the 3000 version which was then loaded on the 3000 and run! The entire porting process took about 2 weeks to complete.

The UNIX operating system perhaps stands at the best testament to C's portability. Most computers which execute UNIX are compatible with each other because they are running the same code. We doubt if this degree of portability has been achieved with any other language.

MAKING RTE SYSTEM CALLS IN HP-UX

Grant Sidwall
Hewlett-Packard (Canada) Ltd.
1825 Inkster Blvd.
Winnipeg, Manitoba R2X 1R3

INTRODUCTION

To create a computerized system, consisting of multiple programs communicating with each other and with I/O devices, a programmer requires capabilities beyond those defined in standard programming languages. These capabilities, such as communication between programs, appropriate scheduling of processes, control of process priority, and I/O to non-standard devices, are normally provided as extensions to the language through subroutine calls to the operating system. The available calls, the capabilities provided, and the concepts implemented are specific to the operating system hosting the application, although the basic set of capabilities and concepts are fairly standard.

This paper takes the basic capabilities with which an HP 1000 RTE programmer is familiar, and shows how the same, or similar, concepts are implemented in HP-UX. The capabilities discussed are: process scheduling, interprocess communication, shared memory, process locking, priority setting, I/O control requests, and asynchronous I/O (class I/O).

PROCESS SCHEDULING

In RTE, a process exists in three areas of memory; an ID segment, a code segment, and a data segment. A new process is created through an EXEC call which causes the operating system to locate free memory for each of the three segments and then copy the appropriate portions of a program file to the segments.

A process in HP-UX also has three main parts; a system data segment, a text (code) segment, and a user data segment. However, in HP-UX, only the initial process at bootup is created from 'scratch.'

All other processes are created by copying an existing process (the *fork* call), and then overlaying the duplicated program with the contents of the desired program file (the *exec* call).

For example, for an existing process to cause a program residing in the file 'SONNY' to be executed, the code in RTE FORTRAN would be:

```
PROGRAM DAD
WRITE (1,('("DAD IS RUNNING")'))
CALL EXEC (10, 6HSONNY ) !assume program is RP'd
WRITE (1,('("SONNY SCHEDULED")'))
END
```

In HP-UX using C, this would be accomplished by:

```
main()
int pid;
{
pid = fork ();
if (pid == 0) /* if I am the new (child) process */
/* overlay myself with 'sonny' */
execl ("/users/grant/sonny", "sonny", 0);
printf ("father is done \n");
}
```

The RTE call causes RTE to locate the program file ('SONNY') via the ID segment, and sufficient memory for the programs code and data segments. RTE then copies the program into memory, including completing the ID segment. (One of the things RTE does in filling out the ID segment is to insert a pointer to the scheduling program so the parent-child relationship can be maintained). Finally, RTE places the program in the schedule list to run at its pre-assigned priority.

The HP-UX process launch is a two step process. The first call (*fork*) causes HP-UX to locate the required free areas in memory (i.e. system data area, user text area and user data area), and then copy the existing process into it. This new process is identical to the calling process in that it is executing at the same point, has its own copy of the same data, and has the same files open at the same points.

The two processes do differ in the value of the process ID (*pid*) returned by the fork call. The value returned to the parent process is the unique process ID number of the child program, while the value returned to the child program is zero.

This difference in the returned value is used to cause only the child process to execute the second step, the 'exec' call. This call causes HP-UX to locate the specified program file and use it to overlay the child process. The process ID of the child process remains the same, so the parent-child relationship is still maintained, but the content of the process (i.e. the program the process is running), is changed. The new program has its own data, starts from its own initial point of execution, may open its own files, and generally bears no resemblance to the process it has overlaid, although it is executing in the same environment (conceptually similar to an RTE SEGLD call).

In RTE, we have the concept of a program being 'busy' (i.e. the process associated with an ID segment exists - the program is non-dormant). This results in RTE having the 'regular' schedule call (EXEC 10) and a queued schedule call (EXEC 24). The regular schedule call will cause the program to be executed only if it is dormant, and will return an error code otherwise. The queued schedule call also causes a program to be scheduled for execution if it is dormant, but, if the program is non-dormant, will cause it to be scheduled as soon as it becomes dormant. (This is made somewhat transparent by the use of the utility routine FmpRunProgram, which creates a new ID segment with a different name if the required one is busy).

Because HP-UX tracks a process by its unique *pid*, rather than by a named ID segment, the concept of a program being unschedulable because it is busy has no meaning. A program can only be requested by the name of the file containing it; a previous process copied from that file has a unique *pid*, and is not related to the new invocation.

RTE also provides the capability for a parent program to wait for a child to finish executing before the parent continues (EXEC 9 or 23). This capability is also provided in HP-UX, but is different in that the parent process may elect to wait for the child at any point in its code, rather than only waiting at the schedule call.

e.g. In RTE:

```
PROGRAM DADWAIT
WRITE (1,('DAD IS RUNNING'))
CALL EXEC (9, 6HSONNY ) !program waits while sonny
                        !runs
WRITE (1,('SON FINISHED'))
END
```

e.g. In HP-UX:

```
main ( )
{
int pid, *statusp;
pid = fork ( );
if (pid == 0)
    execl ("users/grant/sonny", "sonny", 0);
/* parent does something */
printf ("I'm still the parent \n");
wait (statusp);      /*decides to wait for son */
printf ("son is done \n");
}
```

In the HP-UX case, the parent program, after spawning the child, continues execution (the *printf* statement) and only waits for the child to complete (the *wait* statement) when it has no other work to do. Placing the wait statement directly below the *execl* call would achieve the same effect as the RTE EXEC 9 call.

Another feature of program scheduling implemented in RTE is the ability to pass parameters from the parent process to the child, and from the child back to the parent.

Data is passed to the child in the schedule call (EXEC 9, 10, 23 and 24) and received in the child using the RMPAR call to obtain up to 5 sixteen bit words and the GETST or EXEC 14 call to receive a buffer of character data.

This is illustrated in the following parent and child programs (literals are used as much as possible for simplicity):

```

PROGRAM PARENT
INTEGER BUFR(20), PARMS(5), TLOG, I
CALL EXEC(23,6HCHILD ,1986,0,66,7,0,14H,,FOR MY CHILD,7)
CALL RMPAR (PARMS)           !Pick up the 5 integers
CALL GETST (BUFR,20,TLOG)    !Get the returned buffer
WRITE (1,10) PARMS, (BUFR(I),I=1,TLOG)
10  FORMAT (/ " PARMS ARE :",5I6/" STRING IS :",20A2)
END

PROGRAM CHILD
INTEGER BUFR(20), PARMS(5), TLOG, I, SUM
CALL RMPAR (PARMS)           !Get the 5 integers
CALL GETST (BUFR,20,TLOG)    !Pick up character string
WRITE (1,10) PARMS, (BUFR(I),I=1,TLOG)
10  FORMAT ("PARENT SENT 5 INTEGERS :",5I6)/
      "AND 1 STRING :",20A2)
SUM = PARMS (1) + PARMS (3)   !Change
PARMS (1) = SUM               !the
DO 20 I=2,5                   !integer
20  PARMS (I) = 0              !parameters
CALL PRTN (PARMS)             !and return them
CALL EXEC (14,2,12H,,THANKS,DAD,6)
END

```

The result of running the parent programs:

```

C1> PARENT
PARENT SENT 5 INTEGERS : 1986      0   66      7      0
AND 1 STRING: FOR MY CHILD

PARMS ARE : 2052      0      0      0      0
STRING IS : THANKS,DAD

```

CI>

HP-UX also provides the capability for a program to pass data to a child process. However, the data must be in character form, although there is no limit to the number of character strings that can be passed.

The concept is very similar to GETST/EXEC 14 in RTE in that the receiving program expects to pick up a runstring. This is illustrated in the following pair of programs:


```

main ( )      /*the parent program*/
{
  int pid,status;
  pid=fork ()
  if (pid==0)
  execl("users/grant/child","child" "first string",
        "second",0)
  wait (&status)
  printf("Child's exit status was %x \n",status);
}

```

```

main (argc,argv) /*child process expecting parameters*/
int argc; /*parameter count */
char *argv[ ]; /*pointer to array of pointers to
                parameters*/
{
  int i;
  for (i=0;i<argc;i++)
  printf ("Parameter %d is : %s\n",i,argv[i]);
}

```

Running this set of programs produces the following:

```

$ parent
Parameter 0 is : child
Parameter 1 is : first string
Parameter 2 is : second
Child's exit status was 0
$

```

Because the number of parameter strings (*arguments* in HP-UX terminology) is variable, some constructs not found in RTE are used. These are worth explaining.

They syntax of the `execl` scheduling call is defined as:

```

int execl (path,argo,arg1, . . . ,argn,0)
char *path,*arg0,*arg1,. . . ,*argn;

```

The arguments to `exec1` are all pointers to character strings, with the last one being a null pointer (zero) to terminate the list. The first argument, `path`, defines the path to the program file from which the process will be overlaid. The second argument, by convention, is the name of the program file - similar in concept to `GETST` in RTE, which expects to strip out 'RU' and the program name before passing in the 'real' data. The remaining arguments are pointers to strings, each of which is one parameter.

HP-UX determines how many parameters are in the list, and sets the value in `argc` before invoking the child, which is the same as what is done when a program is run interactively.

The child can then pick up the parameters as it needs them; this need not be the first step in the program.

Since the parameters can only be strings, the only way to pass numeric values is to convert them into strings in the parent program, and then convert them back in the child program.

HP-UX does not provide a method of retrieving data from a child program at its termination. Data may be passed back and forth using interprocess communication, which is discussed later, but the only value returned to the wait call is the terminating status of the child program (i.e. the reason it terminated).

INTERPROCESS COMMUNICATION

RTE provides for interprocess communication beyond simple parameter passing at schedule time via the set of EXEC calls referred to as class I/O. This method of communication is based on a key referred to as a class number. This key allows communicating programs to read and write to buffers in the System Available Memory area of RTE. Communication is established by obtaining a class number from RTE (CLRQ call) and passing it to programs which need to communicate with each other.

Communication is accomplished by one program performing a class write (EXEC 20) to the class number to create a buffer containing the desired data, following by another program performing a class get (EXEC 21) to read the data into its area. Note that the class number assigned by RTE through the CLRQ call is essentially random, and may vary each time the program is invoked, requiring that the class number always be distributed to all programs participating in the communication, since any previous value is irrelevant.

HP-UX provides a very comparable service, based on the idea of message queues associated with a message queue identifier. The difference is that the message queue identifier is associated with a user-provided key, so that access to a specific message queue may be requested by specifying a known key.

In other words, if a group of programs agree on a specific key value, either by hard coding it or by deriving it in a systematic fashion, they can access the messages without first requiring some other form of interprocess communication to obtain the key value. In fact, HP-UX provides a routine, 'ftok,' which will return a unique, consistent key value when given a specific file name.

Another feature of message queues in HP-UX is protection. When a message queue is created in HP-UX, its creator provides a set of read/write permissions identical to those for a file (i.e. access is granted in three levels: owner, group, and all other). These permissions can be used to control access to the message queue, and prevent accidental or malicious tampering with the messages.

In some instances, especially in a system being ported from RTE, the programmers may not be interested in the key value associated with the message queue, but simply want a private mailbox as is provided by class I/O in RTE. This is provided in HP-UX by a special case of message queues called *private messages*. To obtain a private message queue identifier, a special key value, referred to as IPC_PRIVATE, is specified when creating the message queue. As in the allocation of a class number in RTE, the message queue identifier returned cannot be determined before it is assigned by the operating system. Consequently, as in RTE, the message queue identifier must be communicated to all programs needing to access the message queue.

In HP-UX, the first 32 bits (a long integer) of each message are treated as a *message type*. This could be considered to be similar to the two tag parameters in RTE class I/O calls in that it is set by the sending program and received by the receiving program. Additionally, a receiving program may specify, through the message type parameter in the message receive call, that it will receive only the type of message specified, or only messages having a type less than or equal to the type specified, or any message regardless of type. In other words, a program can make use of the message type to select specific messages, or ignore it by setting it to zero when receiving.

Messages on HP-UX are implemented in four routines:

- msgget - creates or obtains access to a message queue based on the key value specified.
- msgsnd - send a message to a queue.
- msgrcv - gets a message from a queue (and deletes it from the queue).
- msgctl - performs utility operations such as changing queue permissions, obtaining queue status, and destroying the queue when it is no longer needed.

To illustrate the similarity (and difference) of the implementation, the following sets of programs illustrate the use of interprocess communication to pass a buffer from one program to another for processing and return, first in RTE, then in HP-UX. To keep it simple, error checking and the use of no-wait bits (available in both RTE and HP-UX) are left out.

First, in RTE:

```

        PROGRAM FIRST
        INTEGER BUF(20), CLASS
C      get class number
        CLASS = 0
        CALL CLRQ (1,CLASS)
C      fill the buffer
        DO 10 I=1,20
10      BUFR(I) = I
C      write the buffer to the queue
        CALL EXEC (20,0,BUFR,20,0,0,CLASS)
C

```

```

C      schedule the second program (with wait)
C      and pass it the class number
      CALL EXEC (9,6HSECND ,CLASS)

C
C      read back the buffer and print it
      CALL EXEC (21,CLASS+20000B,BUFR,20)
      WRITE (1,20) BUFR
20    FORMAT (/ "BUFFER IS :",20I3)
C
C      release the class number
      CALL CLRQ (2,CLASS)
      END

      PROGRAM SECND
      INTEGER PARMS(5), CLASS, BUFR(20), TMP
C      get the class number
      CALL RMPAR (PARMS)
      CLASS = PARMS(1)

C
C      read buffer, don't deallocate class #
      CALL EXEC (21,CLASS+20000B,BUFR,20)

C
C      reverse the buffer
      DO 10 I=1,10
      TMP = BUFR(21-I)
      BUFR(21-I) = BUFR(I)
10    BUFR(I) = TMP
C
C      write the buffer back
      CALL EXEC (20,0,BUFR,20,0,0,CLASS)
C
      END

```

Now, in HP-UX using C:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main()                      /* first program */
{
    key_t key;
    int msqid, msgflg, msgsz, i, pid, status;
    struct {
        long mtype;
        short mint[20];
    } msg;
    long msgtyp;
    char msqid_char[12];

```

```

/* get a message queue */
key = IPC_PRIVATE      /* want a private queue */
msgflg = 0600;         /* owner only can read/write */
msqid = msgget (key,msgflg);

/* fill the buffer */
for (i=0; i<20; i++ ) msg.mint[i]=i;

/* write the buffer to the queue */
msgsz = 20*sizeof(short); /* 20 short integers */
msg.mtype = 1L;           /* type must be > 0 */
msgflg = 0;               /* no special features */
status = msgsnd (msqid,&msg,msgsz,msgflg);

/* convert key to characters for passing */
sprintf (msqid_char,"%d",msqid);

/* invoke second program */
pid = fork();
if ( pid == 0 )
    execl ("users/grant/interex/second",
           "second",msqid_char,0);
wait (&status);

/* read the modified buffer back */
msgtyp = 0L;             /* first message on queue */
status = msgrcv (msqid,&msg,msgsz,msgtyp,msgflg);

/* write the buffer out */
printf ("Buffer is :");
for (i=0; i<20; printf (" %d",msg.mint[i++]) );
printf ("\n");

/* release the message queue */
msgctl (msqid,IPC_RMID,&msg):
}

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>      /* second program */
main (argc,argv)
int argc;
char *argv[];

{
int msqid,msgflg, msgsz, i, tmp, status;
struct {
    long mtype;
    short mint[20];

```

```

    } msg;
    long msgtyp;

    /* convert the key from incoming string */
    msqid = atoi (argv[1]); /* second argument */

    /* read in the buffer */
    msgsz = 20*sizeof(short);
    status = msgrcv (msqid,&msg,msgsz,0L,0);

    /* reverse the buffer */
    for ( i=0; i<10; i++ ) {
        tmp = msg.mint[19-i];
        msg.mint[19-i] = msg.mint[i];
        msg.mint[i] = tmp;
    }

    /* send the buffer back */
    msg.mtype=1L;
    status = msgsnd (msqid,&msg,msgsz,0);
}

```

Note that since this example used a private message queue, the message queue identifier had to be passed to the receiving program in the schedule call (as a string, since that is the only form of data that can be passed by that call). If the key had been based on the name of a file known to both (using 'ftok'), the second program could have used *msgget* to obtain a message queue identifier associated with the desired message queue. Also, the message queue *must* be explicitly released when it is no longer needed, or it will remain on the system until manually released using utility *ipcrm*. The status of all active messages can be viewed using utility *ipcs*.

SHARED MEMORY

Another method of communicating between programs is via shared memory, where a portion of memory is mapped into the address space of all participating programs. This is the fastest method of communication between programs, since there is no overhead due to data movement; when data is placed in shared memory, it is immediately available to all participating programs. In RTE, sharable memory is implemented as named common blocks located in the Extended Memory Area.

As such, accessing shared memory requires only a \$EMA declaration in the program code, and an 'SH' directive at LINK time (i.e. there is no active code required in a program to access shared memory). This is very simple, and easy to use, although it does limit the user to only one shared memory segment per program, and that segment is in place of any unshared EMA or VMA.

RTE shared memory can also be locked into memory so that the data in it will be retained even when no active programs are accessing it.

In HP-UX, shared memory is treated as a resource which is acquired by a program as it executes, much like message queues. The process of acquiring access to a shared memory segment is similar to that for accessing a message queue. Access to a shared memory segment identifier is based on a user-specified key value, unless a *private* segment is requested, in which case the system assigns a unique identifier. If a new shared memory segment is being created, the user must specify the segment size in bytes and the file-like permission bits. The segment is then mapped into the users address space, yielding a pointer to the shared memory segment. All access to the shared memory is then done as offsets from this pointer, similar to array addressing in C. A program may attach multiple shared memory segments to itself with each one being considered as a shared array. A program can also detach itself from any shared memory segment to which it is attached, and, by retaining the shared memory segment identifier re-attach to the same segment provided it has not been removed in the intervening time. Shared memory segments can be locked into memory to improve performance in a system where there is heavy paging to disc.

The manipulation of shared memory is accomplished in HP-UX by four routines:

- shmget - creates or grants access to a shared memory segment of user specified size, based on a user specified key.
- shmat - attaches the shared memory segment to the calling program. Returns the address of (pointer to) the start of the segment.
- shmdt - detaches the shared memory segment from the program.

shmctl - performs utility operations, such as obtaining status, changing permission, locking a segment into memory, and removing a segment that is no longer needed.

Because a shared memory segment, like a message queue, can be accessed by any program which knows or discovers the key value or identifier, it is created with the same read and write protection capabilities as a message queue or ordinary file.

RESOURCE SHARING

Once access to a portion of shared memory is obtained, either by RTE's *SHEMA* mechanism or HP-UX's *shmget* and *shmat* calls, it is treated as ordinary memory i.e. all access to shared memory is made through ordinary program language statements, with no operating system intervention to provide synchronization. This can lead to access problems, such as two or more programs writing to shared memory at the same time without knowing of each others activity. This problem is normally handled by an operating system feature which prevents co-operating programs from simultaneously accessing a single resource. The implementation consists of the association of the resource or resources with an entity that has a mechanism to prevent simultaneous use (i.e. guaranteed 'atomic' lock and unlock mechanisms). (The term *atomic* in HP-UX is used to imply an operation that will always complete, once started, before it can be started again by another process). I.e. once one process starts the locking mechanism for a given entity, no other process can start or try to start a lock on that entity until the current activity is completed.

In RTE, the entity associated with the resource(s) is known as a *resource number*, and access to the resources(s) is controlled by *locking* and *unlocking* the resource number. A resource number is allocated, locked, and unlocked using the *RNRQ* call. Because the association between the shared resource and the resource number is made within the co-operating programs, as many or as few resources as the user desires are associated with a single resource number. If multiple resource numbers are used, the locking and unlocking sequences must be carefully established to avoid deadlocks.

In HP-UX, the facility for co-ordinating resource usage is known as a *semaphore*. The process of locking the resource is known as *acquiring* a semaphore, and unlocking is referred to as *releasing* the semaphore.

HP-UX has the ability to deal with a *set* of semaphores (i.e. a group of one or more semaphores, all identified with a single semaphore identifier). All operations and the set are atomic, i.e. if a program tries to acquire (lock) all the members of a set, none of them are locked until all of them are available. This allows programs in which multiple resources are being shared to be coded to avoid deadlock more easily.

The calls used to handle semaphores are:

semget - creates or obtains access to a set of
 semaphores for the supplied key value

semop - acquires and releases semaphores

semctl - obtains semaphore status, sets and resets
 semaphores, and removes semaphores that are no
 longer needed.

In HP-UX, a set of semaphores is assigned read and write permissions when it is created.

An example of three programs communicating through shared memory, and co-ordinating the communication through resource numbers/semaphores follows. In the example, one program initially schedules the two others. The two children write into a 31 element array in memory, which is printed out by the parent program when it is full. The last element in the array is used to keep track of the number of elements written. One child program writes the element number into an array element; the other program writes the square of the element number.

The example is given first in FORTRAN for RTE, then in C for HP-UX:

```
$EMA /SHARE/  
PROGRAM KING  
COMMON /SHARE/ IX(31)  
LOGICAL GOING  
  
C  
  IX(31) = 1      !Initialize counter  
C  
  allocate resource number  
  CALL RNRQ (20B,IRN)
```

```

C      schedule serving programs
CALL EXEC (10,6HSERF2 , IRN)
CALL EXEC (20,6HSERF1 , IRN)

C
C      watch for array to be full
GOING = .TRUE.
DO WHILE (GOING)
    CALL RNRQ (2,IRN)          !lock
    IF ( IX(31).GE.30 ) GOING = .FALSE.
    CALL RNRQ (4,IRN)          !unlock
    DO 12 I = 1,32767          !kill time
12      CONTINUE
    END DO
C
C      write out array
WRITE (1,20) (I,IX(I),I=1,30)
20      FORMAT ("Element ",I2," : ",I3)
C
C      release resource number
CALL RNRQ (40B,IRN)
END

$EMA /SHARE/
PROGRAM SERF1
COMMON /SHARE/ IX(31)
LOGICAL GOING
INTEGER PARMS(5), IRN, COUNT, LOCK, UNLOCK
DATA LOCK /2/, UNLOCK /4/

C
C      get resource #
CALL RMPAR (PARMS)
IRN = PARMS(1)

C
C      change an element every chance we get
GOING = .TRUE.
DO WHILE (GOING)
    CALL RNRQ (LOCK,IRN)
    IF ( IX(31).LE.30 ) THEN
        COUNT = IX(31)
        IX(COUNT) = COUNT
        IX(31) = COUNT + 1
    ELSE
        GOING = .FALSE.
    ENDIF
    CALL RNRQ (UNLOCK,IRN)
    DO 10 I = 1,32767          !kill time
10      CONTINUE
    END DO
END

```

In HP-UX, because shared memory is dynamically acquired, the code is more complex:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
main()                                /*king.c */
{
    key_t key;
    int i,running, semid, shmid, pid, status;
    short *array;
    struct sembuf sops;
    char *shmat();

    /* get a key based on runnable file of this program */
    key = (key_t) ftok ("users/grant/interex/king",'z');

    /* get shared memory segment - 62 bytes (31 short ints) */
    shmid = shmget (key,62,IPC_CREAT|0600);

    /* attach the memory and initialize 31st element */
    array = (short *) shmat (shmid,0,0);
    array[30] = 1; /*elements are 0...30 */

    /* get a set consisting of only one semaphore */
    semid = semget (key,1,IPC_CREAT|0600);

    /* initialize the semaphore */
    sops.sem_num = 0;
    sops.sem_op = 1;
    sops.sem_flg = SEM_UNDO;
    status = semop (semid,&sops,1);

    /* start up the two slaves */
    pid = fork();
    if ( pid == 0)
        execl ("serf1","serf1",0);
    pid = fork();
    if ( pid == 0)
        execl ("serf2","serf2",0);

    /* wait for the array to be full */
    running = 1;
    while (running) {
        /* acquire the semaphore */
        sops.sem_op = -1;
        status = semop (semid,&sops,1);
        if ( array[30] >=30 ) running = 0;
    }
```

```

/* release the semaphore */
sops.sem_op = 1;
status = semop (semid,&sops,1);
}

/* print out the array */
for ( i=0; i<31; i++ ) {
    printf ("Element %d : %d\n",i+1,array[i]);
}

/* destroy the semaphore */
status = semctl (semid,0,IPC_RMID,i);

/* detach and destroy the shared memory segment */
status = shmdt ((int *)array);
status = shmctl (shmid,IPC_RMID,i);

}

```

Note that the HP-UX implementation of semaphores and shared memory offers more capability and flexibility, but requires more user written code.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
main()                                /*serfl.c */
{
    key_t key;
    int i, running, semid, shmid, count, status;
    short *array;
    struct sembuf sops;
    char *shmat();

    /* get a key based on runnable file of king program */
    key = (key_t) ftok ("users/grant/interex/king",'z');

    /* get access to shared memory segment */
    shmid = shmget (key,62,0600);
    /* attach the shared memory */
    array = (short *) shmat (shmid,0,0);

    /* get access to the set of semaphores */
    semid = semget (key,1,0600);
    sops.sem_num = 0;
    sops.sem_flg = SEM_UNDO;

    /* change an element every chance we get */
    running = 1;

```

```

while (running) {
    /* acquire the semaphore */
    sops.sem_op = -1;
    status = semop (semid,&sops,1);
    if ( array[30] <=30) {
        count = array[30];
        array[count-1] = count;
        array[30] = count + 1;
    }
    else
        running = 0;
    /* release the semaphore */
    sops.sem_op = 1;
    status = semop (semid,&sops,1);
    for (i=0;i<32767;i++); /* kill time */
}

/* detach the shared memory segment */
status = shmdt ((int *)array);

}

```

PROCESS MEMORY LOCKING

In an event driven environment, it is often necessary to guarantee that a program will respond to an event in the minimum time possible. One way to accomplish is to have the operating system keep the responding program in memory. This capability is referred to as *program swapping control* or *process locking*. In RTE, this is implemented in the EXEC 22 call which can lock or unlock the data and/or code partitions of the calling program memory. In HP-UX, process locking is implemented by *plock*, which can lock the code and/or data segments into memory, or remove any locks present. The difference between the two is that in RTE you can choose to unlock only the code or only the data partition (or both), while in HP-UX the only unlock option is to remove all locks.

PRIORITY SETTING

In an environment where many processes are running at once, it is often desirable to prioritize the programs (i.e. some programs are more 'important' than others and should get preference).

This differentiation is accomplished by giving each program a *priority* relative to other programs, and constructing the operating system such that programs having a higher priority will get preferential access to resources.

At this point, it is worth considering the way in which RTE handles programs having default priority and programs having equal priority. In RTE, programs of equal priority are treated differently depending on their priority relative to a level referred to as the *timeslice priority*. Programs of equal priority that is lower (higher numerical value) than the timeslice priority are timesliced (i.e. each gets an equal share of processor time). Equal priority programs higher than the timeslice priority take turns running to suspension or completion (i.e. whichever one is running will run until it completes or suspends, then the other will take over). RTE also has a Background Priority setting. Programs with a higher (numerically lower) priority than this value get preferential treatment when decisions are made as to which programs are to be swapped out in favour of other programs.

In RTE, a program's priority is set at compile time to a value in the 'PROGRAM' statement. This value may be overridden at link time, and may be further modified, using the 'PR' command once it has an ID segment. This value may also be adjusted by the program itself, using the *CHNGPR* call, as the program runs. The operating system does not adjust the programs priority in any way according to the load it places on system resources, nor for any other reason. There is no way provided for one program to adjust the priority of another, unless the program goes privileged and writes on another program's ID segment.

Program priority in HP-UX is handled quite differently. In the default method of operation the operating system controls program priority in a way that tries to balance the system load. As a process uses more system resources, its priority is degraded, and as it uses less resources, its priority will improve. This tends to balance the system load in favour of interactive, non-computer bound processes over resource intensive processes so that interactive response time is maintained. One component in calculating a process priority is its *nice* value. This value may be adjusted when a program is invoked - a program with an increased *nice* value has a lower priority and will be less of a load on the system (be nicer, as far as other users are concerned). Special capability is required to decrease a program's *nice* value.

To provide real-time response capability, HP-UX contains a capability to override the normal HP-UX priority manipulations. Using the command *rtprio priority program*, a program may be invoked with an absolute, real-time, nondegrading priority. Such a process executes at a higher priority (lower numerical value) than regular time-share processes. It is, however, timesliced with programs of the same priority, and so will only give up the processor to an equal or higher priority 'real-time' process or certain interrupts. A compute bound, high priority real-time process can effectively halt all other activity, including interactive command processing.

Rtprio can also be used to set the real-time priority of any executing process (*rtprio priority pid*), provided the user has sufficient capability. The above can also be accomplished programmatically using the *rtprio* subroutine.

I/O CONTROL REQUESTS

There are basically three types of I/O control requests:

- i) requests to set device parameters, such as baud rate or magnetic tape density.
- ii) requests to do utility physical operations, such as tape rewind or page eject.
- iii) requests to get the status of the last completed operation on the device.

In RTE, these requests are handled in various ways. Requests to set up configuration parameters are handled by the EXEC 3 call, but these may be temporarily overridden by options in a single read or write request (EXEC 1 or 2). Utility operations in RTE are performed by the EXEC 3 call. Status requests may be handled by an EXEC 3, EXEC 13 or a RMPAR call. In RTE, utility and configuration requests are accomplished by sending the driver a group of settings that the driver stores or passes on to the hardware. Status requests return a structure of status words from the driver's table area.

In HP-UX, all three types of requests are handled through one interface to the operating system, the *ioctl* call. In HP-UX, a read or write request has only three parameters: a file descriptor, a buffer address, and the buffer length.

There is no place for any configuration modification parameters, and the return value is always a transmission log or error flag. Consequently, all configuration changes are handled by *ioctl*, and are deemed to be permanent (i.e. in effect for all subsequent reads and writes. There is an exception to this - when a device file is created, some parameters, such as whether a mag tape should be rewound when its file is closed, are set by bits in the creation command). The general flow of an *ioctl* operation is to read in a structure containing the current configuration, modify the structure, and set the structure back. The modifications are made easier by the availability of *include files* which contain mnemonic names for the various parameter settings as well as definitions of the structures used. This methodology is illustrated by the following subroutine which opens up a serial device file whose name is given as its only parameter, sets the baud rate to 9600, character size to 8 bits, echo on, maps carriage return (CR) to new line (NL) on input, and enables canonical (backspace, etc.) processing.

```
#include <fcntl.h>
#include <termio.h>
#include <sys/ioctl.h>

openterm (dev)                /* routine openterm */
char dev[];
{
  int fd,result;
  struct termio term;         /* parameter structure */

  fd = open (dev,0_RDWR);     /* read/write mode */
  if (fd < 1) return (fd);

  /* get control structure */
  result = ioctl (fd, TCGETA, &term);
  if (result < 0) return (result);

  /* modify the structure */
  term.c_cflag = B9600 + CS8 + CLOCAL + CREAD;
  term.c_lflag | = ECHO + ICANON;
  term.c_iflag | = ICRNL;

  /* send the structure back */
  result = ioctl (fd, TCSETA, &term);
  if (result < 0) return (result);

  return (fd);
}
```

In the case of utility operations, all that may be required is to send the driver a structure describing the operation to be performed.

ASYNCHRONOUS I/O

Asynchronous I/O is non-blocking I/O. That is, it is I/O operation that is initiated by a process, but the process does not wait for it to complete. Instead, the process is free to perform other processing and only completes the I/O request after it has completed some other processing. In fact, if, after the process has done some work, it checks and finds that the request has not completed, the process should be free to do other processing. In other words, the process should be able to try to complete the request without blocking on the completion, as well as not blocking on the initiation.

This facility is provided in a straightforward fashion by the class I/O facility (EXEC 17, 18 and 21) in RTE. Like the interprocess communication facility discussed earlier, the facility is based on buffers linked to a system-assigned class number. An EXEC 17 or 18 call initiates a transfer from or to a device via a buffer placed in SAM. When the request completes, the input buffer or output buffer header is linked to the queue of completed requests for that class. The program can complete the request by issuing an EXEC 21 (class get) on that class number. The class get can be either blocking or non-blocking, depending on the setting of the NO WAIT bit in the call. This provides a very efficient method of doing asynchronous I/O with no polling overhead.

There is no completely similar feature in HP-UX. Similar efficiency and capability can be achieved by using the interprocess message facility to connect the main program to small server programs. The server programs receive messages from one queue and write the messages to their associated device. Data received from the device is written to another queue (possibly using the message type to identify which device wrote it) from which the main program does blocking or non-blocking reads (depending on the NO WAIT flag). HP-UX also includes the *select* statement which allows a program to wait for input from any one of a group of terminals. The wait can be made non-blocking through the use of a timeout, but this differs from the idea of asynchronous I/O in that *select* reacts to any input, not just completed operations, resulting in a process that is more like polling than true asynchronous I/O.

The following program illustrates a single RTE program using class I/O to take data in from two terminals:

```

PROGRAM QUEEN
INTEGER BUFIN(40), CLASS, LU1(2), LU2(2), LU(2)
LOGICAL RUNNING
DATA LU1/71,400B/, LU2/72,400B/, LU/0,400B/

C
C   get class number
CLASS = 0
CALL CLRQ (1,CLASS)
C   prompt users for input
WRITE (LU1,10)
WRITE (LU2,10)
10  FORMAT ("Enter data : ")
C   place class read on each terminal
CALL XLUEX (17,LU1,BUFIN,40,LU1(1),0,CLASS)
CALL XLUEX (17,LU2,BUFIN,40,LU2(1),0,CLASS)
C
NUSERS = 2
RUNNING = .TRUE.
DO WHILE ( RUNNING )
C   wait at class get for input
CALL EXEC (21,CLASS+20000B,BUFIN,40,LUIN)
CALL ABREG (IA,IB)
C   check for termination
IF ( BUFIN(1).EQ.2H!! ) THEN
NUSERS = NUSERS - 1
IF ( NUSERS.EQ.0 ) RUNNING = .FALSE.
ELSE
C   prompt for more input
WRITE (LUIN,10)
LU(1) = LUIN
CALL XLUEX (17,LU,BUFIN,40,LUIN,0,CLASS)
C   write input to console
WRITE (1,20) LUIN, (BUFIN(I),I=1,IB)
20  FORMAT ("Data from LU ",I2," : ",20A2)
C   clear buffer
DO 30 I=1,40
30  BUFIN(I)=2H
ENDIF
END DO
C
C   release class # and quit'
CALL CLRQ (2,CLASS)
END

```

The following pair of programs accomplish the same thing as the previous RTE program, except that in HP-UX we need the server program. The server program performs the same function as the class I/O portion of the RTE system code, i.e. it takes the terminal input and turns it into a message for the main program to receive.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main()                                /*queen.c*/
{
    int midin, midout, pid, status;
    int running, nusers, i;
    key_t key;
    long msgtyp;
    struct {
        long mtype;
        char text[80];
    } msg;

    /* get keys & create 2 message queues (in & out) */
    key = (key_t) ftok ("/users/grant/interex/queen",'i');
    midin = msgget (key,IPC_CREAT|0600);
    key = (key_t) ftok ("/users/grant/interex/queen",'o');
    midout = msgget (key,IPC_CREAT|0600);

    /* start up two servers */
    pid = fork();
    if ( pid == 0 )
        execl ("server","server","/dev/tty03","3",0);
    pid = fork();
    if ( pid == 0)
        execl ("server","server","/dev/tty04","4",0);

    /* send a prompt to each */
    msg.mtype = 3L;                /* for server on tty03 */
    strcpy (msg.text,"Enter data :");
    status = msgsnd (midout,&msg,strlen(msg.text)+1,0);
    msg.mtype = 4L;                /* for server on tty04 */
    strcpy (msg.text,"Enter data :");
    status = msgsnd (midout,&msg,strlen(msg.text)+1,0);

    /* loop for each response received */
    running = 1;
    nusers = 2;
    while ( running ) {
        /* wait for a response */
        status = msgrcv (midin,&msg,80,0L,0);
        /* check for user exit */
    }
}
```

```

    if ( msg.text[0] == '!' ) {
        nusers -=1;
        if ( nusers == 0 ) running = 0;
    }
else {
    /* echo input to console */
    printf ("Data from dev %d : %s",msg.mtype,msg.text);
    /* issue prompt again to mtype in input */
    strcpy (msg.text,"Enter data : ");
    status = msgsnd (midout,&msg,strlen(msg.text)+1,0);
}
}
/* release the message queues */
msgctl (midin,IPC_RMID,&msg);
msgctl (midout,IPC_RMID,&msg);
}

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main (argc,argv)                                /*server.c*/
int argc;
char *argv[];
{
int midin, midout, fd, status, n;
int running, nusers, i;
key_t key;
long mtype;
struct {
    long mtype;
    char text[80];
} msg;

/* open the specified terminal */
fd = openterm (argv[1]);
/* pick up our message type */
mtype = atol (argv[2]);

/* get keys & access to 2 message queues (in & out) */
key = (key_t) ftok ("/users/grant/interex/queen",'o');
midin = msgget (key,0600);
key = (key_t) ftok ("/users/grant/interex/queen",'i');
midout = msgget (key,0600);
/* (in and out are opposite to master) */

/* loop for each prompt received */
running = 1;
while ( running ) {
    /* wait for a prompt */
    status = msgrcv (midin,&msg,80,mtype,0);

```

```

/* echo prompt to terminal */
status = write (fd,msg.text,strlen(msg.text));
/* read input */
n = read (fd,msg.text,80);
msg.text[n]='\0';    /*add string terminator*/
/* pass input up to master */
status = msgsnd (midout,&msg,n+1,0);
/* check for user termination */
if ( msg.text[0] == '!' ) running = 0;
}
}

```

SUMMARY

RTE and HP-UX both provide the user with the tools to build complex multiprogram systems. Generally, the RTE features are implemented in a more specific fashion, i.e. some features use EXEC calls, some use their own subroutines, and some, like shared memory, are implemented transparently. This makes some features easier to understand and implement, while others are more difficult. HP-UX features tend to be more general, and implemented in a more regular fashion i.e. all acquired resources are associated with a user-defined key, and are acquired and released in the same way by similarly named routines. In HP-UX, resources have read and write protection just like files, but they are not automatically deallocated when a program ends, and some, like semaphores, must be manually initialized.

Both RTE and HP-UX are highly capable operating systems. Only someone who is familiar with both systems and the given application can say which would be 'better' for the application, since this will involve trade-offs in development time, system cost, and performance.

INTERFACING HP'S NEW TAPE DRIVES TO HP 1000 A/E/F SERIES

Dave Doxey
Hewlett-Packard Co.
2130 West 2100 South
Salt Lake City, UT 84119

INTRODUCTION

Hewlett-Packard offers a variety of magnetic tape drives for use on HP 1000 computer systems. These include new models as well as those for which obsolescence has been recently announced. It is sometimes difficult to select the tape drive that most closely matches the computer's application, disc capacity, and performance requirements. This paper presents data to aid in this selection process. Particular emphasis is given to the newest tape drives from HP's Greeley Division, the 7974A and the 7978B. Included are actual benchmark data comparing the 7970B, 7970E, 7974A, 7978A, and 7978B on both HP 1000 A Series and E/F Series computers. The advantages and disadvantages of each tape drive are discussed in relation to the application and performance requirements.

HARDWARE

There are a number of possible hardware choices for a 9-track magnetic tape drive on an HP 1000. Some of the tape drives will interface to E/F Series only or the A Series only while others may be interfaced to both. Listed below is a matrix which shows the CPU and tape drive combinations that are supported as of the DSD 4.0 software release:

<u>CPU</u>	<u>DRIVER</u>	<u>MAG TAPE DRIVE</u>
E/F Series	DVR23	7970B/E
RTE-6/VM	DVS23	7974A, 7978A/B*
A Series	DD*23	7970E
RTE-A	DD*24	7974A, 7978A/B

* The 7978A/B will work with the DSD 4.0 RTE-6/VM, but the utilities do not currently support streaming.

The 7970B has been obsoleted by Hewlett-Packard and the 7970E will be obsoleted in the fall of 1986. Obsolescence means that HP will no longer manufacture and sell these products as new. This is a result of technological advances, manufacturing economics, and changes in market demand. The 7970 will still be available as a used or remarketed tape drive. Both hardware and software support will continue for at least five years.

The specifications for the various tape drives are listed below. The 7970B/E is included here and also in the benchmark data because many existing E/F Series users might be contemplating an upgrade to a newer or faster tape drive.

SPECIFICATIONS	7970B	7970E	7974A	7978A/B
Density (cpi) 7974A opt 800	800	1600	1600 1600/800	6250/1600
Approx. Capacity (2400 ft. reel)	20 Mb	40 Mb	40 Mb 40/20 Mb	140/40 Mb
Operating Mode*	S-S	S-S	S-S, Str	Str
Tape Speed				
Start-Stop	45 ips	45 ips	50 ips	N/A
Streaming	N/A	N/A	100 ips	75 ips
Rewind Speed	160 ips	160 ips	200 ips	250 ips
Burst Transfer				
Start-Stop	36 Kb/s	72 Kb/s	80 Kb/s	N/A
Streaming	N/A	N/A	160 Kb/s	468 Kb/s (6250) 120 Kb/s (1600)

* Operating Mode: S-S = Start-Stop; Str = Streaming

A few words of explanation would be appropriate. Tape drives write and read data to and from the tape in physical records. These records may contain just a few bytes of data or several kilobytes. Between each record on the tape, an inter-record gap is written to define the records. This gap is a fixed length, regardless of the size of the data

record. Record size has a significant effect on tape drive performance and tape utilization.

What is a streaming tape drive versus a start-stop drive? Start-stop tape drives have a mechanical tape transport that moves the tape the exact amount necessary to read or write the data that has been requested. The transport may move the tape a fraction of an inch or several inches to satisfy one request. A tension arm usually buffers the tape from the constant "starting" and "stopping" of the transport. The tape movement may seem to be rather jerky to the casual observer. A streaming tape drive attempts to provide a faster, more constant tape movement with the net result of a much higher data throughput. This constant-velocity tape motion is known as "streaming". Data must either be supplied to the drive or accepted by the host computer at a rate fast enough to allow the transport to maintain its streaming operation. If not, the streaming tape drive will operate very poorly, constantly over-shooting the inter-record gap and having to reposition itself for the next read or write. Performance may actually be much lower with a streaming tape drive operating in this manner than a slower start-stop drive.

The 7974A appears to have the best of both worlds, as it can automatically drop into start-stop mode if the data rate slows below that necessary to maintain efficient streaming operation. It also has the option of an 800 cpi density. Thus, a single 7974A with option 800 can replace the functionality of both a 7970B and a 7970E for E/F Series systems.

In the area of high-density (6250 cpi) tape drives, Hewlett-Packard has offered three different models. The 7976A was the first 6250 cpi drive, but it used an expensive vacuum-column transport which also had very high maintenance costs. It was targeted for large HP 3000 systems and was only available to HP 1000 users through purchase of a "special" driver. The 7978A was introduced in early 1984 as a replacement for the high-cost 7976A. The 7978B superseded the 7978A about 18 months later. The only difference between the two is that the 7978B has a 256K data buffer versus a 32K buffer in the 7978A. This buffer size increase has made significant improvements in the performance of most utilities. The tape controller buffers the individual read/write requests until the tape movement can be optimized in an attempt to maintain streaming. This allows margin for operating system overhead and disc accesses while still permitting a streaming operation.

Another advantage to the 7974A and 7978B tape drives is that

they use an HP-IB interface. This means that a single tape drive may be shared among HP 1000, 3000, and 9000 systems. Also, an E/F Series user may now retain his tape drive for use on an A Series if he decides to upgrade his system.

BENCHMARK PROGRAMS

The specifications can give an idea of the performance of a particular tape drive, but they do not take into account operating system and driver overhead. Also, the difference between a streaming and a start-stop tape drive is very significant in actual operation. It was decided to test the different tape drives using various utilities and test programs to provide "real-world" performance data.

First, a test program was written to analyze the raw performance of each tape drive by writing a buffer directly from memory to tape using an EXEC call. The size of the buffer was varied to show its effect on overall performance. This program will also account for the operating system and driver overhead.

The standard RTE backup utilities were tested next. TF, FC, SAVER, and PSAVE were tested for RTE-6/VM, while TF, FC, and ASAVE were tested for RTE-A.

Because of disappointing results with PSAVE and the 7978 on an RTE-6/VM system, a special streaming utility called FSAVE (Fast SAVE) was written to do a track-by-track disc backup. FSAVE is actually composed of two programs, one to read data from the disc and another to write the data to tape. A 49-page SHEMA partition is used to pass the data between the two programs. VMAIO calls are used to read 4 disc tracks at a time into SHEMA. The tape write program then breaks the 4-track buffers into physical tape records of 1 track each. Double-buffering with handshaking is employed so that each program runs at the highest possible speed. This approach minimizes the number of disc accesses and allows a constant flow of data to the tape drive, thus permitting streaming operation at 6250 cpi. FSAVE was written to run on either RTE-6/VM or RTE-A.

All tests were run in a single-user environment with only DS running in background. An F-Series was used for the RTE-6/VM tests and an A900 for RTE-A. Also, all tests were write-to-tape operations only. Read performance should be similar, but it was felt that the most important criteria was the transfer rate for a tape backup rather than a restore.

TEST RESULTS

First, let's look at the performance of each tape drive, giving consideration to operating system and driver overhead. This is the memory to tape test program. This test was performed on a 7978A instead of a 7978B, but the results should be comparable. The difference between the A-series and the F-series performance is slight, thus indicating the similarities in operating system and driver throughput.

Refer to Figures 1 and 2 on page 8. As buffer size is increased, the overall transfer rate in Kbytes/sec increases, but begins to level off at about the 2000 words/buffer mark with the exception of the 7978A at 6250 cpi. The curve is still rising at 340 Kbytes/sec using a 6144-word buffer! This really illustrates the performance of a streaming tape drive at 6250 cpi. You will also notice the difference of start-stop mode versus streaming mode on the 7974A - A900 combination. Streaming mode at 100 inches per second has a transfer rate roughly twice as fast as start-stop mode at 50 ips. Theory proven right again! Also, the 7978A at 1600 cpi is slower than the 7974A streaming because of a 75 ips versus 100 ips difference. All of these performance data seem to illustrate the specification differences of the tape drives. Note, however, that there is some crossover at the 128 to 512 word-per-buffer range. This is due to the inertial and mechanical differences of the drives. Streaming mode definitely suffers when you can't get enough data to the tape drive to maintain the transport in motion at the proper speed.

Now let's see what happens when we add a disc access to the process. These next tests were performed using the standard RTE utilities and the custom-written FSAVE. A disc seek and read is required to read in a buffer before it can be written out to tape.

ASAVE is able to take advantage of immediate-completion I/O calls available in RTE-A, thus we see the streaming capability at 6250 cpi. RTE-6/VM does not have this option, so in order to allow a streaming backup utility, a two-program, double-buffering scheme must be used as described in the benchmark section.

The transfer rate for these disc backup utilities is a function of both tape record size and disc access method. TF, FC, and SAVER use tape record sizes in the 2K-word range, while WRITT, PSAVE, ASAVE, and FSAVE typically write an entire disc track in a single tape record. This record

size is at least 4K words and often 6K words. Disc access method has perhaps the greatest effect on performance. TF, FC, and SAVER are file backup utilities and must search the directories and open the individual files, thus incurring the overhead of the file system. There can be several disc accesses involved and possibly many if the files have extents. There are even differences between TF and the other file utilities. FC and SAVER do all directory searches prior to saving any files to tape, while TF does its directory searches as it saves each file. The flexibility of TF comes with a penalty in performance. The physical backup utilities do not have to access directories or files. They operate on a track-by-track basis for an entire disc LU or unit.

These differences are evident in Figures 3 and 4 on page 9. Note the significant difference in performance of FC on RTE-6/VM and RTE-A. FC on RTE-6 allows a much higher throughput with the streaming tape drives than does RTE-A because of the way the driver DVS23 operates versus DD*24. RTE-6's DVS23 allows a default streaming mode for all subsequent I/O requests while DD*24 requires that the streaming bit be set in each individual EXEC call. FC and TF do not set the required streaming bit for each EXEC call, thus the performance is severely degraded on RTE-A. ASAVE does not have this handicap, however, and similar performance is seen for both ASAVE/FSAVE on RTE-A and FSAVE on RTE-6/VM. PSAVE's performance suffers because of its internal operation. PSAVE goes the extra mile to preserve data integrity by doing two separate disc reads for each track and then comparing the checksums. These extra operations cause the overall throughput to be reduced below that required for streaming. If a PSAVE verify is also invoked, the performance suffers even more.

CONCLUSIONS

Now that the results have been compiled, which is the right tape drive to choose? For most HP 1000 systems with disc capacities of 404 Mb or less, the 7974A would be the best choice. It is less expensive than the older 7970E, and offers better performance both in start-stop and streaming modes. It is supported by both RTE-6/VM and RTE-A and can "grow" with the system when an upgrade is required. Also, the 800 cpi compatibility option is important for some applications. For systems with larger disc capacities (and there are some big ones!), the 7978B would be justified because of the savings in both tape media and backup time when operating at a density of 6250 cpi. Physical utilities would be the most efficient means of backing up large

amounts of disc storage with occasional incremental file backups using TF.

Existing HP 1000 systems could also benefit from the addition of a new streaming tape drive to complement a 7970. A 7978B could be used for high-speed physical backups while the 7970 would best fit the role for incremental file backups.

FUTURE DIRECTIONS

Hewlett-Packard is committed to designing, manufacturing, and marketing products that fill a need and make technological contributions. The current offering of 7974A and 7978B drives meets these criteria. This family of streaming mag tape drives will very likely be expanded in the future.

Perhaps the best news of all is that Data Systems Division is committed to provide software support and utilities for the Greeley Division tape drives.* This includes support for RTE-6/VM running on E/F Series machines. The software driver DVS23 is now a standard driver rather than a "special". A boot loader ROM is also available for use with a version of !BCKOF which supports the 7974A and the 7978B tape drives. A new file backup utility will soon be released on both RTE-A and RTE-6/VM which will support streaming at 6250 cpi. A streaming physical backup utility for RTE-6/VM will also be released. There really is a future for HP 1000 users!

F-SERIES MEMORY TO TAPE

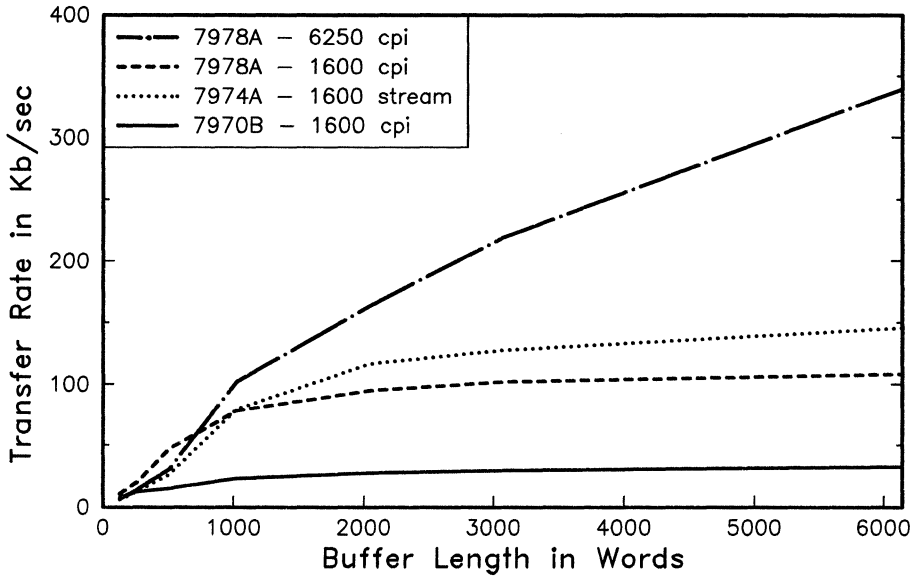


Figure 1

A900 MEMORY TO TAPE

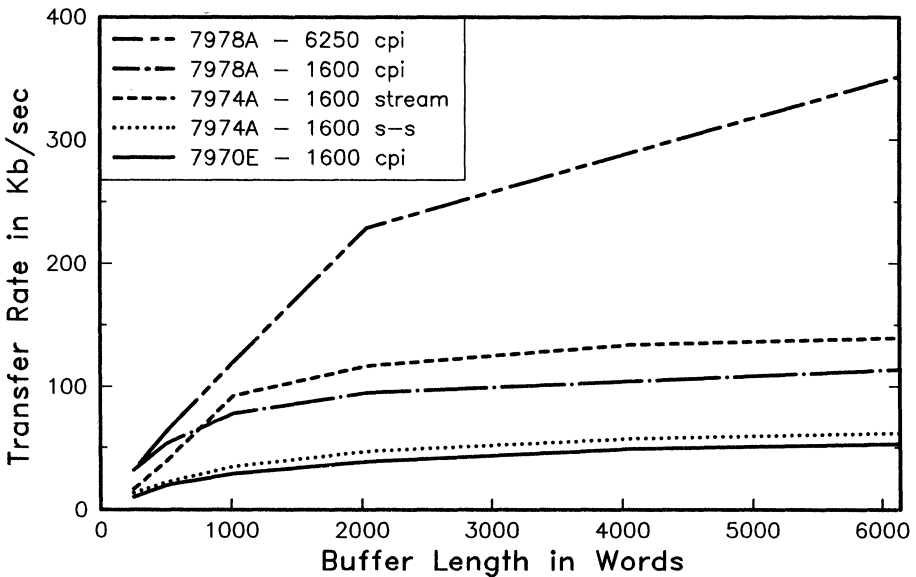


Figure 2

TAPE BACKUP ON RTE-6

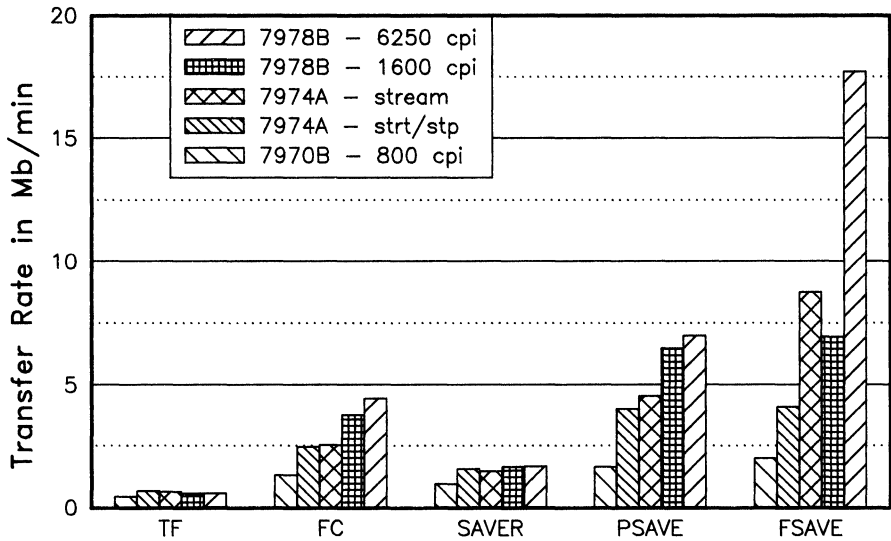


Figure 3

TAPE BACKUP ON RTE-A

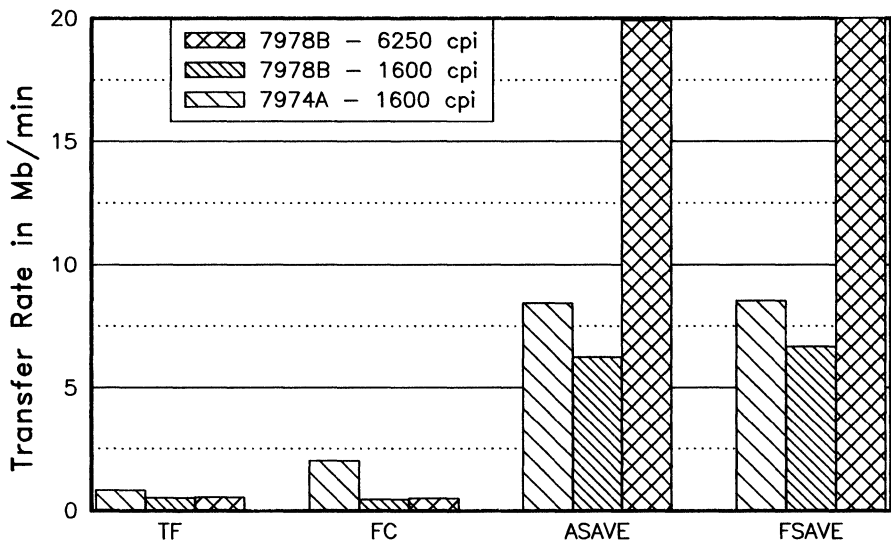


Figure 4

HP-UX: Using Standards to Solve Real World Problems

Val Jermoluk and Chris Bego
Hewlett Packard/mailstop 42LV
19420 Homestead Road
Cupertino, CA 95014

Introduction

It was only seven years ago that UNIX* systems first arrived in the commercial market. Since then, their importance has been hotly debated. Proponents enthusiastically insisted benefits such as hardware independence, a wealth of applications software, and portability would lead to spiraling growth rates and eventual domination of the multiuser market. However, by 1984 the UNIX operating system held only a small share of the commercial computing market. The "wealth" of UNIX software began to be referred to as a myth, especially outside of the technical arena. Would the market for UNIX products ever mature?

In 1985, several events clarified the future of the controversial operating system. AT&T published a specification for a standard implementation of the UNIX operating system, the System V Interface Definition, ensuring applications portability between conforming UNIX systems. Many of the largest computer users including the U. S. government and General Motors began to require the UNIX operating system for new acquisitions. A critical mass of applications software appeared, evidenced by fat third party catalogs from AT&T and /usr/group, market data, and growing commercial activity at industry tradeshowes such as UNIX EXPO and UNIFORM. Major computer vendors such as IBM, Amdahl and DEC, gave their stamp of approval with stronger commitments to UNIX products. Finally, the UNIX operating system has become a major force in the multi-user computing market.

This paper will highlight HP-UX, HP's implementation of the UNIX operating system and its unique contributions to the marketplace. First of all, though, why has the UNIX operating system become so important?

Benefits of the UNIX operating system

The attention showered upon the UNIX operating system is well deserved. Its feature set and position as a standard offer the customer several benefits not achieved with other operating systems. A UNIX user realizes:

- portability of software between UNIX systems
- hardware and vendor independence
- easy access to existing applications software
- multi-vendor networking

* UNIX is a registered trademark of AT&T

- lower software development costs
- early access to new technologies

These benefits stem from the revolutionary design and composition of the operating system itself. AT&T Bell Labs originally developed the UNIX operating system in 1969 for internal use. A few years later, it was rewritten in C, a high level language, making it much easier to port from machine to machine than operating systems written in assembly language. The modular structure of the UNIX operating system also facilitated porting, as only a small portion is customized to the hardware. AT&T originally made UNIX source available to universities for a small fee and offered it to the public in the early 1980's. At that point, hardware vendors had an inexpensive and readily available alternative to developing a new operating system from scratch. This alternative has proven popular: today UNIX systems are available from over 74 vendors (Infocorp, February 1986).

Moving applications between machines with identical operating system interfaces is straightforward, requiring only recompilation and relinking on the new machine. Portability has been a major plus for UNIX from the start, but as the number of implementations has grown, variations of the operating system have cropped up, hampering the ease of porting applications. To stem the divergence of the UNIX operating system, AT&T published the first System V Interface Definition (SVID) in 1985. The SVID clearly defines a machine independent UNIX operating system interface. It focuses on the interface to application software, not on the implementation. Adherence to the SVID by implementors of UNIX systems guarantees portability of applications with any similarly compliant systems. The SVID has quickly established itself as a defacto industry standard with acceptance by all major UNIX system vendors. Even proponents of the BSD versions developed at U.C. Berkeley are jumping on the SVID bandwagon: Sun Microsystems has signed an agreement with AT&T to converge their 4.2BSD-based operating system with the SVID. The SVID is thus ensuring one of the biggest advantages to owning a UNIX system, the ease of porting software.

HP's Software Evaluation and Migration Center (SEMC) can testify to the portability of SVID compliant software. They recently ported HILO-3 from Genrad (a logic simulation package consisting of 150,000 lines of C code) to HP-UX in 4 days. Minx, an inventory and control package with 170,000 lines of C code, was ported in 4 hours. Compare this to months or years when porting to proprietary systems!

As the UNIX industry has matured, the number of software applications have multiplied. Today there are over 600 applications listed in the 1986 product published by /usr/grp. AT&T lists over 500 applications supported on System V. Because this software is portable, it is accessible to all UNIX users. HP's catalog lists over 270 software packages currently supported on HP-UX and this number is growing rapidly.

Since the UNIX operating system resides on a diverse assortment of machines ranging from personal computers like the IBM PC to supercomputers such as the Cray-II, users with UNIX software are no longer dependent on a particular hardware or vendor. This freedom means access to a range of hardware and a protected software investment ... vendor independence.

The widespread acceptance of the UNIX operating system has indirectly created new defacto standards in the area of networking. The TCP/IP protocol and ArpaNet services developed at U.C. Berkeley have become a defacto standard for Local Area Networking on UNIX systems. The Network File System (NFS) developed by Sun Microsystems delivers transparent file access among computers from many different vendors. With their latest release of System V.3, AT&T introduced RFS, a transparent file access system designed for UNIX machines only. While RFS has yet to prove itself in the market, TCP/IP with Arpa/Berkeley Services and, to a lesser extent, NFS are widely implemented defacto standards. HP's Series 300 and Series 800 computers support TCP/IP with Arpa/Berkeley Services, allowing HP-UX users to communicate in a multi-vendor network.

The UNIX operating system was developed by program developers expressly FOR program development. Today, almost twenty years later, it is still regarded as the premier software development environment. Programmers, armed with over two hundred utilities and the ability to connect them become more efficient. Finding programmers already proficient with the UNIX operating system is easy due to its prevalence in university curriculum. The combination of less training and increased programmer productivity drives software development costs down.

The owners of UNIX systems are privileged with early access to new innovations in hardware and software because the UNIX environment provides scientists an existing, well understood and flexible base. Development work with RISC architecture, artificial intelligence, parallel processing, fourth generation languages, etc., is being done on UNIX systems.

Perceived drawbacks of the UNIX operating system

As you see, the UNIX operating system offers invaluable advantages. However, because it was designed for use in a cooperative, timesharing development environment, some of its most exciting characteristics have been seen as a drawback in other markets. Features which have hindered wider acceptance include disc caching, program scheduling, the kernel preemption scheme, the user interface and security. Let me review each for you, and later discuss HP's solution.

The UNIX operating system speeds disc access in a manner transparent to the user through "disc caching". Frequently used portions of the disc are kept in memory, minimizing the number of timeconsuming disc accesses. Disc caching is a popular performance booster but when a

power outage occurs and the contents of memory are lost, the effects can be debilitating. Why? With a disc cached system, many files and information on the structure of the file system are kept in memory, allowing for potential loss of data and corruption of the file system. At the January UNIFORM trade show (the largest exhibition devoted to UNIX systems) half of the floor lost power temporarily. All of the affected systems went down, and some didn't come up for many hours after the power resumed. HP's solution to this problem for our customers desiring high data integrity will be discussed later.

Operating system schedulers determine when each independent task, or process, will be run by the cpu. Tuned for a timesharing load, the UNIX scheduler allocates the available computing resources fairly, giving each process a proportionate amount. The user can assign higher priorities to critical tasks, but the scheduler dynamically adjusts the priorities of all processes to ensure each gets equal treatment. Such a scheduler is inadequate for time critical tasks requiring an immediate response. HP's creation of a more autocratic or deterministic scheduler for real-time applications is addressed later.

The speed with which a process can get the attention of the cpu may also be critical. With UNIX systems, all operating system functions (such as disc I/O and system calls) have priority over any user job. The kernel will not "give up the CPU" until the current service completes, or is blocked. During this time, the system will respond to interrupts by entering I/O drivers, but will not execute a realtime program. Real-time tasks, by definition require a predictable and immediate response and must have the ability to preempt or interrupt any other task. Without kernel preemption traditional UNIX systems have stayed out of the real-time arena. Later HP's unique innovations in this area will be examined.

One of the most contested aspects of UNIX is its ease-of-use. For experienced application developers, it is the system of choice, unsurpassed in providing the power and flexibility needed for software development. On the other hand, the user interface, full of terse yet powerful commands, is frightening to novices and casual users who rightfully complain of a steep learning curve, cryptic command names, and a confusing array of options. (You can tell its creators were hunt-and-peck typists with "cc" standing for C compiler, "mv" for move, etc. Do you know what the command "grep" does? It stands for "global regular expression printer".) To take full advantage of the power of the operating system the user must understand the concepts of recursion, iteration, regular expressions, etc., - in other words, have considerable programming expertise. System administration can be overwhelming to the novice as well, accomplished by over 50 individual commands which assume a great deal of knowledge and experience on the part of the system administrator. Clearly, traditional UNIX systems are a far cry from the friendly menus, icons, etc. of today's personal computers.

The UNIX operating system was designed for use in a small, cooperative environment. As a result, system security - the ability to prevent unauthorized access to certain information - was a low priority. The powerful development tools enable experienced programmers to easily access the internal structure of the system and gain information outside of their assigned jurisdiction. A sophisticated system administrator with a thorough understanding of the UNIX operating system is required to build increased levels of protection. The U.S. government has addressed the issue of operating system security head on by defining strict guidelines for classifying systems called the "Trusted Computer System Evaluation Criteria". HP is currently evaluating how to best meet their requirements with HP-UX.

HP-UX: Meeting standards and going beyond ...

Weighing the growing market opportunities for UNIX systems and the unique advantages it offers our customers against any drawbacks, HP has chosen to sell a UNIX operating system on a range of computers. "HP-UX", as we call it, offers all the advantages of a standard UNIX operating system but goes beyond other implementations to overcome the drawbacks for our target markets. Some of the unique contributions of HP-UX which set it apart from other operating systems - without compromising on compatibility - include:

- powerfail recovery
- real-time capabilities
- enhanced user interfaces
- the Device I/O Library
- Native Language Support

HP-UX on the Series 800 Model 840 protects customers from data loss during a power failure. First, the programmer can select whether or not the disc cache is flushed to the disc at the end of a set of I/O requests. This control of data flow to and from the disc minimizes the possibility of data loss from external power failure. Secondly, the Model 840 can suffer a power failure for up to 15 minutes in a 24 Mb system and then recover gracefully when power is returned. User programs continue where they left off, not losing data. HP-UX offers the added performance of disc caching without the drawbacks.

Modifications to the scheduler in HP-UX are just one set of enhancements ensuring control for realtime applications. Unlike traditional UNIX systems, HP-UX allows the programmer to:

- * assign real-time processes a non-degrading, higher priority than any other process
- * have a real-time process preempt any process with a lower priority
- * have a real-time process take over the cpu, even while it is in the midst of an operating system function (impossible on traditional UNIX systems). This is called "kernel preemption".

Not only is HP's kernel preemption unique, but it is fully transparent to users and application programs. (Available on the Model 840 only.)

The end result of these and other enhancements is a UNIX system on the Model 840 with real time performance roughly equivalent to an established industry leader, the HP 1000 A900.

HP-UX products offer a choice of interfaces for users with different needs. HP Windows/9000 allows the user to create and move windows to house applications (including graphics) either interactively or programmatically. Also supported is the creation of custom menus, icons and type fonts. HP Windows provides a highly productive working environment as well as a valuable component for solution creators to customize for their applications. The Personal Application Manager, or PAM, gives the user simple access to powerful applications and operating system functions via a friendly and easy to use menu. AXE or the Application Execution Environment provides a low cost, compact HP-UX environment on the Series 300 for the user desiring only to run applications. (The commands and utilities needed to create, compile and modify software are not available.) AXE incorporates PAM, HP Windows and other features to allow the user to use and maintain the system without knowing HP-UX. For example, users can pick an application they want to run from a menu on the screen. HP Windows, PAM and AXE are just three alternatives to the standard user interface.

HP's experience in the instrument business has been incorporated into HP-UX through the Device I/O Library or DIL. DIL makes programming HP-IB and parallel I/O simpler and more powerful. It allows the user/programmer to easily read from, write to, and control HP-IB and parallel interfaces to connect unsupported peripherals.

HP leads the UNIX industry in helping customers provide international solutions. Native Language Support, or NLS, provides the tools for an programmer to produce applications which are adaptable for use in different countries or local environments. An enduser of such an application could, for example, enter data from a keyboard laid out in his local alphabet, have it stored unchanged, sort it according to the ordering of his alphabet, read error messages in his own language, and print it out in the local character set. A single application can be modified to run in another language without recompiling or modifying the source code. NLS also allows many HP-UX commands to send error messages in the local language, assisting the programmer as well as the enduser. Today there are 18 supported languages including Japanese, Turkish, and most European languages.

A broad, compatible family of products

The powerful features of HP-UX are available on an impressive range of hardware.

At the low end is the Integral PC, a single-user, multi-tasking HP-UX computer system in a fully integrated, 25 lb. transportable package. The IPC is easy-to-use: it turns on and off like a personal computer and is automatically configured at power on.

The HP Series 300 workstations provide a wide choice of power and performance through their modular, plug-in design. The Series 300's are based on the industry available MC68010 and MC68020 micro processors.

The Series 500 multi-user 32-bit computers support up to 64 users and are ideal for high-performance graphics, computation or general purpose applications.

The Series 800 Model 840, tops the HP-UX family today as HP's first superminicomputer based on the Precision Architecture. It provides fast computational power for specialized tasks within engineering, CIM, scientific and general technical applications.

As you can see, the HP 9000 family provides a range of power wide enough to address many computing needs, linked together with a common operating system. To optimize the potential of this family, corporate-wide goals have been established to ensure that all HP-UX products support:

- 1) effortless applications and user migration from AT&T UNIX, and easy migration from other popular UNIX environments
- 2) effortless applications and user migrations between any HP-UX systems
- 3) added value to encourage migration to HP-UX

A closer look at the composition of HP-UX reveals our strategy for meeting these goals. At the heart of HP-UX lies AT&T's System V Interface Definition, Issue 1 (SVID). In the SVID, AT&T clearly defines a version of the UNIX operating system which acts identically on different hardware and is machine independent. Software designed to run on one SVID compliant system will run without change on any other. The SVID has quickly been adopted by all major vendors including Sun, DEC, Microsoft (Xenix), Amdahl, Data General, Apollo, Sperry, NCR, etc., etc. By complying to the SVID, HP-UX provides applications portability within the HP 9000 family and with other SVID compliant systems.

In addition to the SVID, HP-UX contains almost all of the hardware independent features of AT&T's System V.2. HP-UX also contains enhancements developed at U.C. Berkely and HP. These have been incorporated into our internal standard to ensure consistency across the HP 9000 family. Through adherence to industry standards our customers realize the benefits of hardware independence, portable software and a very secure software investment.

HP-UX: Key to HP's operating system strategy

No other major computer vendor has as strong a commitment to a UNIX operating system as HP. In fact, HP-UX is HP's primary operating system targetted for the design, engineering, factory automation and computation intensive markets. How does this fit into HP's overall operating system strategy?

The MPE operating system available on HP 3000 computers has been designed and tuned for what we at HP call the "commercial market". This includes transaction-based applications such as financial accounting, manufacturing planning and control (e.g. MRP and MM), and office automation. The HP 1000 and HP 9000 have been targetted for applications in factory automation, design engineering and process control.

With the increasing demand for UNIX solutions from all markets, the traditional boundary line between "technical" computers and "commercial" computers is blurring. In fact, customers are beginning to request HP-UX products for traditionally commercial applications. While the HP 3000 with MPE is still our main thrust for these applications, HP is monitoring customer demand closely to determine if we should incorporate special features into HP-UX for on-line transaction processing applications.

Now that full real-time functionality has been achieved with HP-UX on the HP 9000 Model 840, the distinction between the HP 1000 and HP 9000 families is less clear. A comparison of real-time performance between the A900 and the Model 840 shows them to be about equal: where the Model 840 is a bit faster at CPU bound real-time tasks, the A900 is a bit faster for I/O bound tasks. These differences plus their disparate processing capability and price make them easy to position. For most real-time applications, especially those requiring dedicated processors, a low cost solution, or optimal I/O performance, the HP 1000 family provides the best fit. The HP 1000's can coexist with the Model 840 on a network or use the Model 840 as an upward growth path. Migration from RTE to HP-UX is facilitated with HP's PORT/HP-UX package.

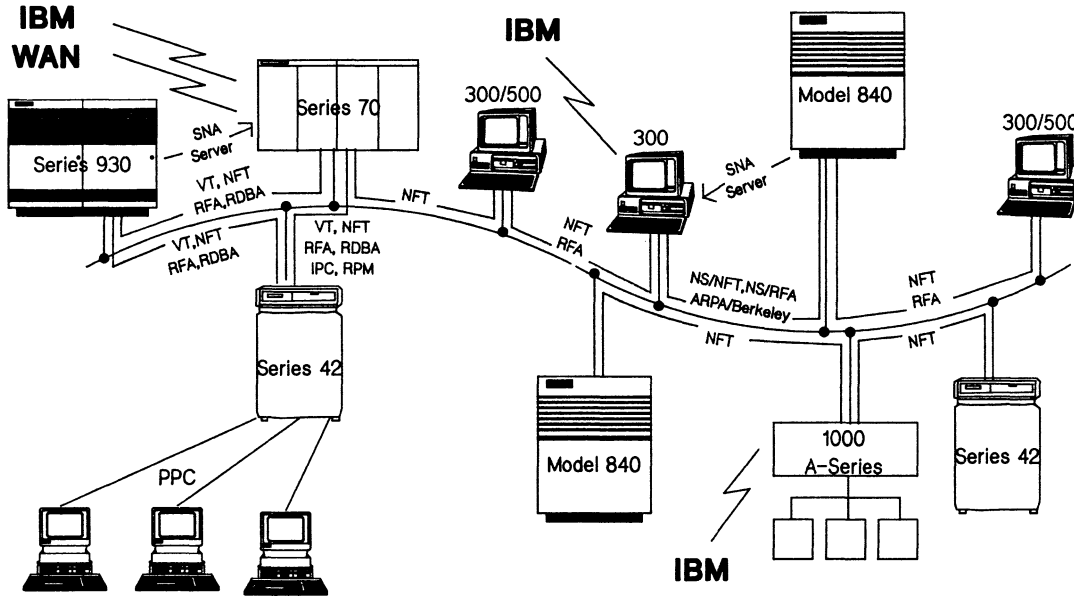
Essential to HP's overriding goal of providing complete solutions is the integration and coexistence of all of our products. Through our networking strategy, HP-UX, MPE, and RTE products can transfer files with each other, access systems from other vendors such as IBM and DEC, etc. to solve a multitude of problems. (see figures)

HP representatives work actively with industry standards committees on the evolving definition of the UNIX operating system.

Our efforts in the area of real-time UNIX have paid off - not only does the Model 840 have most of the base real-time features likely to be required by General Motors and other industry leaders - but

HP-UX: Using Standards to Solve Real World Problems

THE DATACOMMUNICATION SERVICES THAT EXIST TODAY



NOTE: Products either exist or will be orderable by December, 1986.

VT	= virtual terminal	RDBA	= remote database access
NFT	= network file transfer	IPC	= Interprocess communication
RFA	= remote file access	RPM	= remote process management

fmdrw2

SUMTDY

(Orderable by December 1986)

HP System-to-System Networking Matrix

(Orderable by December 1986)

Services

DDA
LLA

Link Level Access or Direct Driver Access

	NFT RFA HP-UQ <i>ARPA/4.2BSD</i> NetPC 802.3 Ethernet	NFT RFA (HP-UQ) IPC 802.3 Ethernet	NFT RFA (HP-UQ) ARPA/BSD4.2 802.3 Ethernet	NFT RFA (HP-UQ) ARPA/BSD4.2 802.3 Ethernet	NFT RFA (HP-UQ) LLA Ethernet
HP 9000 S/800					
HP 9000 S/500					
HP 9000 S/300					
HP 9000 S/200					
HP 1000 A Series					
HP 1000 MEF Series					
HP 3000 37-70					
HP 3000 930 950					
Vectra					
to VAX					
to IBM					

* Thru HP 9000 Series 300 server S/800

HP 9000 S/800 S/500 S/300 S/200 HP 1000 A Series HP 1000 MEF Series HP 3000 37-70 HP 3000 930,950 Vectra

Services in italicized are not yet available will be orderable by

*Services in italicized print
are not yet available, but
will be orderable by 12/86.*

the key real-time extensions developed by HP are expected to comprise most of the spec in the first-round of IEEE standards

/usr/group consists of representatives from many companies with UNIX concerns and works to define standards in the areas of internationalization, database, graphics, networking, real time, performance, and security. HP representatives participate in all of these. One project manager co-chairs the internationalization group.

HP's active involvement in IEEE's P1003 standards group for UNIX includes co-chairmanship. The group has just published their first trial-use standard for a machine independent implementation of UNIX, P1003.1. Although AT&T's SVID is widely accepted today as the base standard for UNIX systems, many companies would like to follow a publicly controlled standard such as P1003.1 if it were fully defined. The National Bureau of Standards is considering using it as a basis of a federal standard, ANSI is considering adopting it, as is ISO.

HP is involved in close, cooperative relationships with international standards groups such as X/OPEN, the UNIX consortium in Europe.

HP's active involvement in standards committees demonstrates our commitment to working in the public interest. By staying abreast of the latest developments, we can promptly respond to new standards. Standards bodies also provide a forum for HP's proprietary developments to be considered for wider adoption.

Future directions for HP-UX

Hewlett Packard is committed to tracking industry standards in the future. Specifically, we are evaluating how to incorporate AT&T's latest release, System V.3. Providing distributed systems is also critical to our strategy, and investigations are underway to determine the best approach. Both NFS and RFS are under consideration. In the area of windowing, X has emerged out of MIT as a defacto standard. Future windowing products from HP will be based on X and maintain source code compatibility to HP Windows/9000. (HP Windows/9000 will continue to be supported to provide full object code compatibility to existing Windows/9000 applications.) With X, users will realize performance improvements and greater portability of windowing software. HP has recently begun a partnership relationship with DEC and MIT to steer the course of X. To make custom, friendly user interfaces for HP-UX as easy as possible to design, a set of interactive development tools are being created.

A version of the UNIX operating system created by Microsoft explicitly for use on personal computers, Xenix, has enjoyed great popularity. In fact, Xenix sits on over 52% of all UNIX systems (IDC Yates), and supports over 500 commercially available software applications. HP is working with Santa Cruz Operation to port Xenix

5.0 to the Vectra PC. The product will be introduced later this fall and is targetted for sales through the OEM/VAR channel to small businesses. How do Xenix and HP-UX relate? They are both based on the SVID, so applications are portable across the two. However, each has its own set of extensions: HP-UX is tuned for engineering and manufacturing applications, and Xenix is tuned for the personal computer user in a business environment. Providing a migration path between HP-UX and Xenix is under investigation. Come to the UNIX Expo tradeshow next month in Manhattan for a first hand view!

We have touched on just a few directions HP-UX will be taking. We are committed to providing standard products with the necessary features to solve our customers problems. Keep in touch with us and our sales force to let us know if we are on track!

Decreasing Realtime Process Dispatch Latency Through Kernel Preemption

David C. Lennert

Hewlett-Packard Company
19447 Pruneridge Avenue
Cupertino, CA 95014
hplabs!hpda!davel

ABSTRACT

A key measure of a realtime system is how quickly a waiting process can be dispatched in response to some event (for example, I/O completion). One major component of this is the time it takes to preempt the currently executing process. In a traditional UNIX† system, a process executing in user code can be preempted immediately. However, when executing in the kernel, the process gives up the CPU only voluntarily and explicitly (for example, by blocking for some unavailable resource or by completing a system call). The kernel can therefore execute for a significant period of time before giving up the processor to another process. This period of time is called preemption latency and, when significant, it is unacceptable in a realtime system. This paper describes modifications to the HP-UX kernel which substantially reduce this time. Measurement results are presented which quantify these times and the improvements that have been made.

1. INTRODUCTION

This paper discusses one aspect of realtime operating system performance, process preemption latency, and presents changes made in the HP-UX operating system for the HP9000 Model 840 which reduce this latency.

First a brief overview of realtime system concerns and features is given. Then the process preemption latency problem is defined and alternative solutions are discussed. Finally an overview of the chosen solution and its implementation are presented followed by measurement results which quantify the improvement made.

2. REALTIME SYSTEMS

2.1. Realtime system concerns

A realtime operating system distinguishes itself from a non-realtime operating system in that it responds to a real world event within a real world time constraint. This is usually defined in terms of *event response time* and/or *sustained data throughput*.

Quick event response time allows a process to respond to an event fast enough to satisfy some external requirement. One example would be repositioning a cursor in response to a moving mouse fast enough to appear instantaneous to a human observer. Another example would be halting the supply of steel to an automated assembly line when a jam occurs.

† UNIX is a trademark of AT&T.

A large sustained data throughput allows quickly generated data to be gathered. If the process cannot gather data as fast as the external source is generating it, then data loss usually results. An example would be digitizing a map.

Note that the performance requirements imposed on realtime systems usually come from the physical world. The cost of not meeting the requirements can be minor (screen jitter), major (hip deep in steel), or catastrophic (reactor meltdown). This variation in cost gives rise to a range of demands on the reliability of realtime performance (from "most of the time" to "all of the time").

2.2. Realtime system features

To address these performance needs realtime systems employ features which are usually absent from non-realtime systems.

Process priorities determine the importance of a process so that a more important (stronger priority) process will execute before a less important (weaker priority) process. Timeshare systems typically adjust a process' priority frequently while it runs. On UNIX systems a process can change from a stronger to a weaker priority with respect to another process (or vice versa) many times a second. *Non-degrading priorities* allow a process to maintain a fixed (usually stronger) priority with respect to other processes and thus obtain a constant, maximum preference.

Another factor which delays process execution is swapping or paging. If a process is swapped or paged out of memory when it needs to run then response latency is increased. Allowing *processes locked in memory* prevents this problem.

Processes usually wait while a requested I/O operation (for example, a device read) is performed. Higher data throughput can be obtained by allowing a process to overlap execution with its I/O operations. This *asynchronous I/O* capability is sometimes provided directly by the operating system or can be implemented via multiple processes communicating through a high speed mechanism such as shared memory synchronized with semaphores.

3. PREEMPTION LATENCY

3.1. The problem

Non-degrading priorities and memory locked processes give maximum preference to a realtime process which is ready to execute. There is, however, one major factor to overcome before the process can execute: If another process is currently executing then it must be preempted and the realtime process restarted.

Traditional UNIX systems can preempt a process immediately while the process is executing in user mode. If, however, the current process is executing within the kernel then it only voluntarily and explicitly gives up the processor. Specifically, it gives up the processor when either 1) the kernel calls the `sleep()` routine to suspend the current process until a needed resource becomes available, or 2) the kernel returns control to the user program at the completion of a system call thereby returning to user mode and allowing preemption to occur. The kernel can execute for a significant period of time before giving up the processor to another process. This greatly increases process preemption latency and decreases the system's ability to provide quick and predictable event response time.

3.2. Alternative solutions

The goal is to decrease the amount of time the kernel executes before it gives up the processor to a waiting higher priority realtime process. To achieve this goal there are two basic alternatives: 1) the kernel can be made to execute all of its functions more quickly or 2) the kernel can be made to tolerate interrupting its execution in deference to the waiting process (preemption).

The former is clearly a superior approach as it has the side benefit of causing the entire system to execute faster and with less kernel overhead. It can be achieved through a combination of faster hardware and algorithmic changes. In addition to algorithm changes which reduce total execution time one can shift code from the kernel into the user program. One example would be to implement the file system manipulation code in a user library and leave only the code supporting basic device access in the kernel. This allows more of the "kernel" to execute in user mode where it is readily preemptable. The problems with moving kernel algorithms into user mode are a loss of reliable security checking and a loss of atomicity of operation with respect to other processes. In addition, there is only so much kernel code that can be reasonably moved into user mode. In the final analysis, preemption latency is typically left unacceptably high. So the second alternative, increasing the preemptability of the kernel, is explored.

The problem with making the kernel arbitrarily preemptable is a loss of atomicity. Kernel data structures can be viewed as memory which is shared among all the user processes. Each process makes requests of the kernel which update this shared data. There must be a mechanism which ensures that these updates are performed atomically; otherwise, faulty operations and a system crash usually result. Simultaneous (multi-processor) and interleaved (uni-processor) data structure access is prevented either through one or more semaphores (which reduces the problem to updating shared semaphores) or by preventing even the possibility of contending access. The latter approach is usually provided by atomic hardware instructions and/or so architecting the system that such colliding accesses never happen.

The mechanism used in a traditional UNIX system is this latter approach: nothing interrupts a process while it is running in the kernel. (The exception to this, I/O interrupt processing, will be discussed later.) This implementation has too coarse a granularity. That is, the data structure "lock" can be held for a long period of time which can prevent other processes from running even if they don't access the same data structures being currently updated. Thus the lock covers more data structures and lasts for a longer period of time than is usually needed. It is this drawback which gives rise to the poor preemption latency of the UNIX system.

4. SOLUTION IMPLEMENTATION

4.1. Overview

The preferred solution is to use multiple semaphores and have each semaphore control access to an independently used data structure. No other process will access the data structures which the preempted process is using since no other process has the necessary semaphores locked. The kernel can then be immediately preempted at any point in its execution. This results in the fastest preemption time but requires that the entire kernel be modified to adhere to semaphoring conventions. Just sorting through the various data structures and assigning semaphores can be a large amount of work. This approach is typically employed in multi-processor systems. For a description of one such implementation and the effort required see [Bach84] and [Felton84].

An approach which is easier to implement is to find places in the kernel where it is already safe to preempt and only allow preemption there. (Such a "safe place" is a spot or region in kernel code where all kernel data structures are either updated and consistent or locked via semaphore.) This does not require modifying the entire kernel to conform to a new data access philosophy. It does have several drawbacks though. Rather than occurring immediately, preemption is held off until the next "safe place". Also, our experience has shown that these safe places are not found but made. It is easier, however, to make a "few" safe places than to rewrite a kernel.

Because implementation schedule was of strong importance, our solution combines both these preemption styles: there is a synchronous method which allows preemption at a specific point during kernel execution, and an asynchronous method which allows preemption

anywhere during a region of kernel execution.

4.2. Details

The synchronous method is useful when places can be identified in the kernel where data structures are either in a consistent state (i.e., between an access transaction) or all required resources are locked via some semaphoring mechanism.

The synchronous method is invoked by placing a call to the macro `KPREEMPTPOINT()` at such a safe place in the kernel. This macro merely checks a global flag, `reqkpreempt`, which indicates the presence of a higher priority realtime process which is ready to run and calls a function, `kpreempt()`, to cause a `swtch()` to the process. (The `reqkpreempt` flag is similar in function to the `runrun` flag used in typical UNIX systems to indicate that a higher priority timeshare process is ready to run.)

There is also a function variant of `KPREEMPTPOINT()` called `IFKPREEMPTPOINT()` which returns true if a pending preemption was serviced; otherwise it returns false. This is useful if lengthy algorithms need to allow preemption but, if preemption occurs, there are assumptions which may have been invalidated and now must be rechecked.

The asynchronous method is useful when preemption can be tolerated over a region of execution and synchronous polling via `KPREEMPTPOINT()` would incur unacceptable overhead (for example, large memory copies during fork, exec, or user I/O). This method is implemented via a software-generated interrupt which is recognized by hardware. Hardware causes an asynchronous transfer of control to the trap handling routine (similar to a pagefault taken inside the kernel when accessing user pages) which in turn calls `kpreempt()` to preempt the kernel.

Hardware recognition of this interrupt is controlled via `spl` levels. The routines `splpreemptok()` and `splnopreempt()` have been added to allow and disallow recognition, respectively. Both of these new `spl` levels are weaker than other `spl` levels (i.e., they do not hold off other interrupts). Also, other `spl` levels are stronger and hence imply that kernel preemption is held off. Thus, interrupt processing activity which typically runs at `spl4()` or higher automatically disables preemption. Usually the kernel runs at `splnopreempt()` (whereas before kernel preemption it used to run at `spl0()`, which no longer exists).

When a higher priority realtime process becomes runnable, kernel preemption is requested by calling `preemptkernel()`. This routine sets the `reqkpreempt` flag and generates the hardware supported interrupt. The flag and interrupt are both cleared by `swtch()` whenever it switches to a new (highest priority) process.

The now pending preemption request is serviced at the first point when either:

- a) a `KPREEMPTPOINT()` is executed (which tests the `reqkpreempt` flag),
- b) the `spl` level drops to `splpreemptok()` (which allows the pending interrupt),
- c) user mode is entered (this is just one case where the `spl` level drops to `splpreemptok()`), or
- d) `swtch()` is called (which always transfers to the highest priority runnable process).

In total, approximately 180 synchronous preemption points and 20 asynchronous preemption regions were added to the HP-UX kernel.

4.3. Limitations

There is one overriding limitation on what kernel preemption can accomplish: Kernel preemption can only preempt (suspend via `swtch()`) an operation which is being executed within a process context. It cannot preempt interrupt processing code and allow a process to execute because the UNIX system does not support this type of operation.

Therefore, all interrupt processing is implicitly considered to be of higher priority than any (realtime) process. This means that no matter how quickly preemptable one makes the

kernel, if interrupt processing becomes unacceptably time consuming then timely kernel preemption cannot be achieved. So the only option is to reduce interrupt processing overhead to an acceptable level.

Note that, even if all individual interrupt servicing operations are short, kernel preemption can be held off for an arbitrarily long time by many quickly arriving (back-to-back) interrupts during heavy I/O activity. There is nothing that can be done in this situation since interrupt processing, by definition, has priority over all process execution.

In addition to the typical I/O driver code, the UNIX system allows non-I/O code to be executed in an interrupt processing context. This facility, called the *callout queue*, causes a kernel procedure to be executed at a specified time offset. The procedure is invoked from an interrupt processing context during clock interrupt servicing. This is usually done at a weaker interrupt priority than all other I/O interrupts. (See [Stra86] for a more detailed discussion of the callout queue mechanism.)

4.4. Overcomming limitations

To minimize callout queue execution overhead, a separate system process was created to provide a preemptable process context in which to execute some lengthy callout queue code. This process, the *statdaemon*, is a lightweight kernel process similar to the scheduling daemon or the pageout daemon. It waits for the *lightning bolt event*[†] and then executes a standard set of statistics gathering routines. These routines represent the lengthy portion of the *schedcpu()* function. (Among other things *schedcpu()* recomputes process priorities every second; see [Stra86] for a discussion of its operation.) A new routine, *sendlbolt()*, is now scheduled on the callout queue in place of *schedcpu()*. *Sendlbolt()* performs the quick functions of *schedcpu()* including generating the lightning bolt event.

4.5. General performance improvements

In addition to the preemption specific modifications, kernel preemption times were improved by several general performance improvements. These include making the process table multi-threaded and placing entries in different states on different lists, as well as using hashing techniques to speed data structure searches. See [Feder84] or [McKusick85] for a discussion of similar improvements.

4.6. Debug facilities

A set of debug facilities allow dynamic control over the preemption system. These facilities are conditionally compiled into the kernel and include the ability to turn preemption on or off, enable kernel preemption for timeshare as well as realtime processes, and forcing (almost) all preemption points to "preempt" (sleep for a random amount of time).

4.7. Timing measurement facilities

In order to tell how long the kernel executes without blocking or preempting, and where in the kernel these long execution paths are, the kernel was instrumented to collect timing measurements.

Timing measurements are taken by sampling the time at kernel entry and exit (*syscall()*, *trap()*, and *swtch()*) and also whenever the kernel changed between a preemptable and non-preemptable state (*KPREEMPTPOINT()* and all *spl* routines). The time intervals during which the kernel executes in a non-preemptable state are logged. Also, in order to tell where in the kernel this execution occurs, a program counter (*pc*) procedure call trace of the kernel stack is collected when the time is sampled, and the *pc* traces for start and stop points of the interval are logged along with the delta time.

[†] The lightning bolt event is a standard UNIX event which occurs frequently, for example, every second.

However these times fluctuate based on the amount of interrupt processing activity which occurs during the measurements (since device interrupts are injected "randomly" into non-interrupt kernel execution). To account for this, the measurement system also counts the interrupt processing time which occurs during a kernel path measurement and logs this as well. This allows the interrupt processing time to be precisely removed from the "normal" kernel execution times. Results are reported both ways.

The interrupt processing time is counted in a fashion similar to the "normal" execution time. Whenever the processor switches to or from the interrupt stack the time is sampled. On exiting the interrupt stack, the time difference is computed and added into an ongoing count. This count is zeroed at the beginning and logged at the end of a "normal" measurement.

In order to obtain sufficiently accurate times, a new kernel routine was introduced to obtain clock time accurate to a microsecond.

Data logging is done into a circular kernel buffer. A user level program retrieves the data by reading the pseudo-device file, `/dev/kmem`, which accesses kernel memory.

A workload is run on a kernel equipped with the measurement system while data is collected into a file. The file is later reduced by a set of user level programs.

5. MEASURED IMPROVEMENT

5.1. Presentation of results

To determine the improvement made in realtime process dispatch time, two sets of measurements were taken using the timing measurement facilities discussed above. One set with kernel preemption enabled and one set with it disabled. The same workload was run during both measurements.

The workload consisted of a suite of tests which validate the correct working of all kernel functions. Because the instrumented kernel precisely measures each section of the kernel every time it is executed, it is not necessary to execute a kernel code path more than once.

The raw results consist of a stream of times. Each time represents an interval when the kernel was non-preemptable. As one way of summarizing the results, a distribution was computed which represents the percentage of total kernel execution time that was spent in code paths of less than x milliseconds.

Figure 1 shows this distribution for both preemption on and preemption off. Figure 2 shows a blowup of the portion of Figure 1 near the y axis and more clearly shows the "preemption on" case. Each graph can be interpreted as: y percent of kernel execution time was spent in code paths shorter than x milliseconds, or y percent of the time, the kernel can be preempted in less than x milliseconds. A sharply rising curve indicates that more of the total system time was spent in quickly preemptable code paths. It is readily apparent that without kernel preemption most of the kernel execution time occurs in code paths which are fairly long. The endpoint on each curve indicates the maximum observed value (observed worst case) for this test run. Note, however, that the curve labelled "Without Kernel Preemption" has been clipped in Figure 2; the real endpoint is shown in Figure 1.

The results in Figures 1 and 2 are summarized in Table 1.

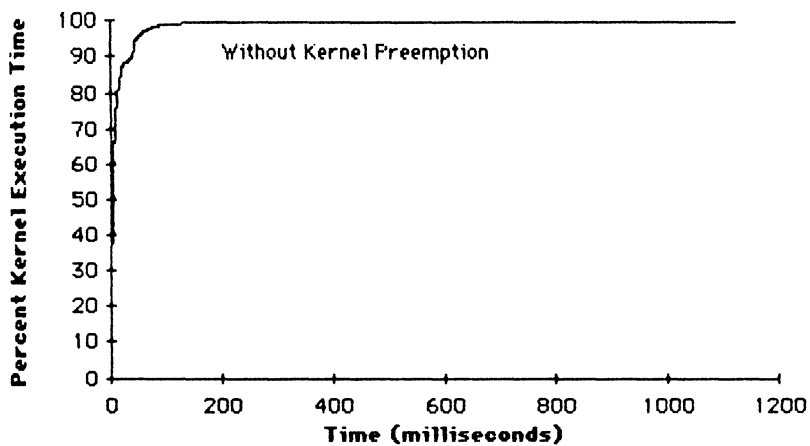


Figure 1

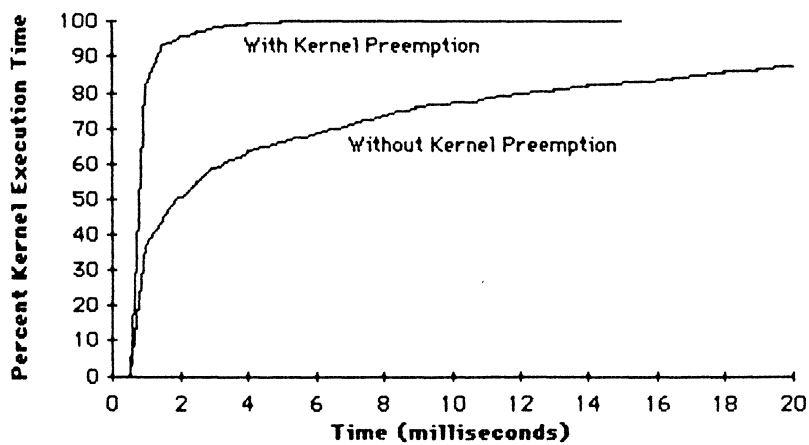


Figure 2

Table 1 Non-preemptable Kernel Time (milliseconds)			
	Preemption Off	Preemption On	Improvement
90% kernel	40 ms	1.4 ms	x28
99% kernel	129 ms	3.4 ms	x37
max kernel	1127 ms	14.6 ms	x77

Disclaimer: Note that these results are for a particular run of a particular workload. Results will vary from run to run and from workload to workload.

5.2. Discussion of results

It was found that a traditional UNIX system (equivalent to measurements made with kernel preemption disabled) could provide acceptable realtime process dispatch time in some cases but it could not provide it consistently (and hence not reliably). This is due to the fact that, while many code paths in the UNIX kernel are short, some code paths can execute for a very significant amount of time. Unfortunately, this problem is not limited to only those kernel code paths executed on behalf of the realtime application; any simultaneously running application(s) could cause the kernel to execute a lengthy code path. This means that, on a traditional UNIX system, a realtime application which works simultaneously with one background workload may fail with a different workload or even with the same workload on a different occasion.

In contrast, Table 1 shows that HP-UX kernel preemption has provided significant improvements in realtime process dispatch time. In the worst case observed with our workloads the improvement was well over 50 fold. With preemption enabled it was found that non-preemptable kernel code paths were significantly shorter, and more consistently so, than in the traditional case. This resulted in better and more reliable timely dispatch of realtime processes regardless of background workload.

In the case where preemption is off, the longest non-preemptable code paths are typically large data copies during process creation (fork), process overlay (exec), or user I/O operations. Other major contenders include process termination (exit), shared memory setup operations, and file link/unlink. In the case where preemption is on, the current longest code paths are now in the terminal driver.

These results represent the status of HP's preemption tuning activities as of this writing; work is currently underway to further reduce these times.

6. EVALUATION AND FUTURE DIRECTIONS

The time to dispatch a waiting process in response to an external event has been made significantly smaller and more uniformly predictable. This has allowed the HP-UX system to solidly address the performance needs of a much broader range of realtime applications.

Future work will entail further improvements to realtime performance via increased kernel semaphoring in order to address more stringent application needs.

7. ACKNOWLEDGEMENTS

The following people from Hewlett-Packard contributed to this work: James O. Hays introduced preemption points and regions into the memory and process management systems; he also offloaded interrupt processing functions into the newly created statdaemon. Sol F. Kavy introduced preemption points and regions into the file system. Suzanne M. Doughty edited early versions of this paper.

8. REFERENCES

- [Bach84] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems" *AT&T Bell Lab. Tech. J.*, **63**, No. 8 (October 1984), pp. 1733-1749.
- [Feder84] J. Feder, "The Evolution of UNIX System Performance", *AT&T Bell Lab. Tech. J.*, **63**, No. 8 (October 1984), pp. 1791-1814.
- [Felton84] W. A. Felton, et. al., "A UNIX System Implementation for System/370" *AT&T Bell Lab. Tech. J.*, **63**, No. 8 (October 1984), pp. 1751-1767.
- [McKusick85] M. Kirk McKusick and Mike Karels, "Performance Improvements and Functional Enhancements in 4.3BSD", *USENIX Conference Proceedings*, Summer 1985, pp. 519-531.
- [Stra86] Jeffrey H. Straathof, Ashok K. Thareja, Ashok K. Agrawala, "UNIX Scheduling for Large Systems", *USENIX Conference Proceedings*, Winter 1986, pp. 111-139.

Interpreters/Compilers - Their Differences and Merits

Husni Sayed
IEM, Inc.
P.O. Box 8915
Fort Collins CO 80525

INTRODUCTION

The use of computers by Engineers and Scientists is rapidly increasing. Programming is becoming a requirement in the field, due to the nature and the complexity of the problems faced by Engineers and Scientists on a daily basis.

The problems solved by Engineers and Scientists can be broken down into 4 major categories: experimentation, modeling, process control, and CAD/CAM.

Experimentation is carried out in the lab: the Engineer or Scientist is experimenting with a new body of knowledge. The job of the Engineer or Scientist in this instance is to come up with new algorithms or techniques to fit each experiment. Computer programs may be used to control instruments or devices in the lab.

Modeling entails taking a predetermined algorithm, and putting it to work on specific data. The role of the computer in this situation is to vary inputs to an algorithm, and monitor the outputs. A model "simulates" a situation to see how a system will behave under a given set of circumstances.

Process Control programs are used mainly in a manufacturing process, and sometimes in the lab. This is similar to modeling in that the computer uses a pre-existing algorithm to measure inputs and outputs. Process Control, however, is guided more toward the manufacturing process. Such programs usually require continuous feedback to the computer: adjustments to the process are made automatically as needed.

CAD/CAM programs are used for a wide variety of industrial and scientific applications. CAD/CAM applications tools aid in the design and manufacturing of products and product components.

Problems requiring modeling, process control, or CAD/CAM are fairly static problems, with static algorithms. The user takes a predetermined program, and monitors or varies inputs and outputs. Static problems such as these require little programming, as the necessary algorithms are already available to the user, and change very little.

Experimentation, on the other hand, is the area that may require greater programming skills. As the problem is not always well-defined, an algorithm must grow and change as the user's understanding of the problem grows and changes. Dynamic algorithms are constantly undergoing modification.

This paper is going to focus on the differences between interpreters and compilers, and the advantages and disadvantages of each in the solving of dynamic problems.

COMPILERS VS. INTERPRETERS

Most Engineers and Scientists are not programmers, but they need some programming skills to stay ahead of the complex problems they are trying to solve. The biggest problem in this area is that software tools are not advancing as quickly as hardware. It is essential that Engineers and Scientists have friendly and tolerant systems available for their use. They need to spend more time using their machines, and less time fighting them.

An Ideal System for Engineers and Scientists

The Ideal system for Engineers and Scientists requires:

- support for structured programming and modularity, to facilitate development of easily maintained code;
- support for library development and maintenance, so an algorithm can be stored and used in other programs;
- support for single-statement matrix operations (such as vector operations, evaluation of determinants, or initialization of an identity matrix), so that commonly used operations are performed easily, without the need to create an entire algorithm;
- support for a rich set of math functions, and;
- support for graphics and data acquisition, so that massive amounts of data can be analyzed and transformed into visual results.

A system meeting all these requirements would need to:

- be tolerant toward human programming errors;
- have built-in error detection and recovery routines that could be accessed from within a program;
- have friendly and powerful debugging tools;
- allow for incremental programming (the ability to make quick changes and see the results), and;
- produce results very quickly once the program is developed.

Tools Currently Available

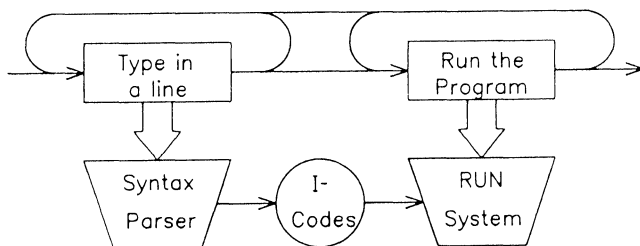
Currently, an Engineer or Scientist can choose either a compiler or an interpreter to aid in program development.

A *compiler* is a computer program that translates another computer program (one that is written in an English-like language) into the machine language bit pattern. The resultant "machine code" can be understood and carried out directly by the CPU of the machine.

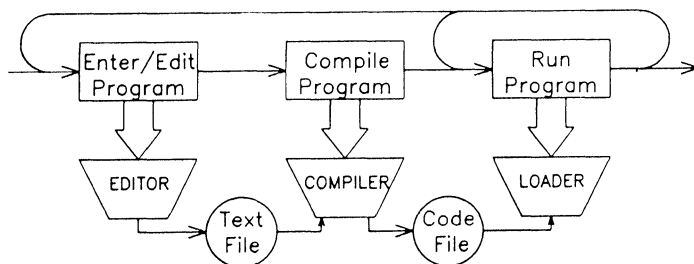
An *interpreter* is a computer program that can directly execute the tasks specified by another program (which is represented in an English-like language).

The Program Development Cycle

The program development cycle in an interpreted environment is a continuous, inter-related process. There are no definite steps involved in the cycle. A programmer can type in one or more lines, executing those lines at any time, and modify the lines, or type in more lines, etc. So the "unit of program development" is the line: programs are executed on a line-by-line basis:



In a compiled environment, however, there is a definite distinction in the steps of program development:



and the different steps do not interact at all. The "unit of program development" is an entire program. A program must be syntactically complete before it can be compiled, and it must be compiled before it can be run.

Compiled VS. Interpreted Memory Usage

INTERPRETER

The BASIC interpreter produces I-codes, which are the interpreted intermediate code produced when a line is typed in. I-codes are a "compact" duplication of your source code. At run time, the interpreter makes use of three main data structures: the symbol table, the value area, and the BASIC stack.

The symbol table keeps information about what variables are used in the program, lines addressed by GOTOs or branching, etc.

The value area is the area where the values of elements defined in the symbol table are kept. When you operate on a particular variable, its name is in the symbol table, but its value is in the value area.

The BASIC stack is the mechanism which controls the execution techniques used by the interpreter. The interpreter operates on a reverse Polish notation technique: it pushes two operands onto the stack, and then performs the operation on them.

COMPILER

On the same operating system (the BASIC Operating System), the compiler produces machine language instructions as the representation of program source code. The compiler also makes use of a value area (there is no symbol table), and three control mechanisms: the system registers, the system stack, and the BASIC stack.

The system registers are the most efficient, and are used to perform simple operations, and pointer address operations.

The system stack is used to store intermediate results of expression evaluation. It is not as efficient as the system registers, but is more efficient than the BASIC stack.

The BASIC stack is the least efficient mechanism used. It is needed to call operating system routines.

Interpreted VS. Compiled Code

In an interpreted BASIC program, each BASIC line is represented as a series of "intermediate codes" (referred to as I-codes):

line header	I-code	I-code	...	EOL
----------------	--------	--------	-----	-----

Each I-code represents an activity that must be carried out by the interpreter. The EOL represents the end-of-line I-code, and it occupies one byte. The line header occupies 6-8 bytes, depending upon whether or not the line is labeled.

An interpreted BASIC program is fairly compact as far as code size goes: the I-codes do not occupy too much space. The "expense" of interpreted BASIC occurs at run time. Each I-code has to be fetched and recognized by the interpreter. After this, an interpretation of the I-code semantics is made, and the I-code instruction is carried out.

In a compiled program, every BASIC line is represented directly in machine code. Machine code is a form that can be understood by the computer: it is a direct, exact representation of the semantics of the original BASIC line. Since the statements have been "expanded" into machine code, a compiled program will be larger than an interpreted one. Even though the file is larger, a compiled program usually executes faster than the interpreted version: this is because the overhead associated with fetching and recognizing instructions at run time is eliminated.

COMPUTE-BOUND CODE

Compute-bound code is code that consists of mainly computational statements. Computational statements include assignment statements, arithmetic (+, -, *, /), etc. Compute bound code tends to produce favorable results when compiled.

In an interpreted BASIC program, the expression "A = B + C" is represented as:

line header	12	13	14	232	144	EOL
----------------	----	----	----	-----	-----	-----

where the I-code:

- 12 tells the computer to push the address of A onto the BASIC stack
- 13 tells the computer to push the address of B onto the BASIC stack
- 14 tells the computer to push the address of C onto the BASIC stack
- 232 tells the computer to add B and C, and push the result on the stack
- 144 tells the computer to store the value on the top of the stack in A

These I-codes occupy about 5 bytes of storage. The interpreter will execute at least 200 machine code instructions to carry out the operation.

In a compiled program, the statement "A = B + C" is represented in machine code as:

```
move.w    -26(A4), D0
add.w     -28(A4), D0
move.w    D0, -24(A4)
```

This machine code occupies 12 bytes. However, this operation is executed by 3 machine code instructions.

For this assignment statement, the *code expansion ratio* (the amount the code expands when compiled) is 12/5, or 2.4. This means that the compiled code is about 2.4 times the size of the interpreted code. It should, however, be about 40 to 70 times faster.

I/O BOUND CODE

I/O bound code is code that consists mainly of I/O or graphics operations. This would include PRINT, READ, etc. I/O bound code does not fare as well as compute bound code when compiled.

In interpreted BASIC, the statement "PRINT A" is represented as:

line				
header	155	0	164	EOL

where the I-code:

```
155  initializes the print operation
0    tells the computer to push the address of A onto the BASIC stack
164  performs an I/O operation
```

These I-codes occupy about 3 bytes.

In a compiled program, the statement "PRINT A" is represented in machine code as:

```
lea      global_pntr, A5
move.l   A2, B_TOS
jsr      Print_init
movea.l  Valptr, A4
movea.l  B_TOS, A2
movem.l  -8(A4), D0_D1
movem.l  D0_D1, -(A2)
clr.w    -(A2)
movea.w  #443, A3
move.l   A3, IPC
movea.l  36(A4), A0
jsr      Goto_OS
```

This machine code occupies 64 bytes.

The code expansion ratio for this example is 21. The form shown above is one of the worst case examples for compiled BASIC. On the average, the code expansion ratio for I/O bound code will be between 2.5 and 4.0. The code expansion ratio for a PRINT statement with 20 items after it would be much smaller: the compiled program has no additional overhead after the first item is printed. With an interpreted program, the overhead would continue for each item in the list to be printed.

The speed increases for I/O bound code are not as great as for compute bound code. On the average, the speed increases that can be expected for I/O bound code are from 20% to 50% (1.2 to 1.5 times faster), depending upon the device and operating system routines being used.

Advantages/Disadvantages of Compilers VS. Interpreters

In general, an interpreter has the advantage at program development time, and the compiler has the advantage at run time. The advantages of an interpreter include:

- Quick feedback during program development. The programmer can type in a few lines, run them, and get results. This lends itself to incremental programming.
- Small code size. User programs are very compact because of the use of I-codes.
- Powerful system and error control. Events, keys and interrupts are very easy to control from within an interpreted environment.
- Symbolic debugging and easy line tracing. Because the symbol table is always present at run time, symbolic debugging is trivial. Programmers can retrieve the values of variables at any time, parse the program, modify them, and resume the program.
- Source available during program execution. The programmer can always refer back to the source code.
- Quick and easy changes can be made to the program.

The disadvantages of a interpreter are:

- Slow execution time. The fetching and translating of I-codes at run time causes a considerable delay at execution time.
- Program source is unprotected. Because the program source code is always resident in memory at run time, it cannot be hidden from other users.
- Portability to other systems is limited.
- More memory is needed at run time to hold the symbol table.
- Redundant parsing at run time. Every time the program is run, each line must be re-parsed and re-translated.
- Comments are expensive, because they occupy space in a program, which is resident in memory at run time.
- Repetitive constants are expensive. Interpreted BASIC does not support a constant pool to hold constants used in a program.

The advantages of a compiler include:

- Fast execution time. Compilers are almost always faster than interpreters. The difference in speed will vary, depending upon how intelligent the compiler is, and how well it presents the code. All tasks of syntactic parsing are performed at compile time, so translation is only done once.
- Program source is protected. The source code need not be resident at run time, and the original program is very difficult to reconstruct from the object code.
- Portability to other systems is only limited by the hardware and operating system capabilities.
- Symbol table is not resident at run time, therefore less memory is occupied.
- System control is bound only by the availability of operating system tools. Any capabilities supported by the operating system, and available to the compiler writer, can also be supported in a compiled environment.
- Constant pool eliminates redundancy. A constant is not stored more than once in a context that is compiled.

The disadvantages of a compiler are:

- Slow program development. The user must wait until compilation is complete to see the results of a program. Compilation may be seconds or minutes, depending upon the size and complexity of a program.
- Larger code size. In a compiled program, the code to perform a task is expanded at compile time. Therefore, a compiled program occupies more space than the I-codes produced by the interpreter.
- Symbolic debugging requires a resident debugger. To debug a program that is already compiled, the user needs a debugger that can control the execution, and the symbol table must be STORED into memory.
- Programs are not easily modified. Since every modification to the program requires editing, compiling, and running, quick changes to the program cannot be made.

CONSIDERATIONS

MEMORY USAGE

In the program development cycle, an interpreted program takes about the same amount of memory as a compiled one. In the program execution stage, a compiled program generally occupies more RAM than the equivalent interpreted program.

PROGRAM SPEED

In the program development stage, the interpreter is usually faster than the compiler, because the user can type in a line, and see the results immediately. In the program execution stage, however, the compiler is definitely faster than the interpreter.

CONCLUSION

The ideal system for Engineers and Scientists would combine the ease of interpreted program development with the speed of compiled program execution. The best solution is a system that can make use of both an interpreted and a compiled language. With such a system, the user could develop and test programs in an interpreted environment, and once a program is fully developed, compile it to achieve the speed increases at run time. In such a case, the time spent to compile a program is not a major factor, as compilation need only occur once.

DISC PERFORMANCE ON HPUX

Carol A. Hubecka
Hewlett-Packard Company
Disc Memory Division
P.O. Box 39
Boise ID 83707

INTRODUCTION

This is a discussion of disc performance on HPUX; Hewlett-Packard's version of the UNIX* operating system. The paper is divided into two parts. The first half is a discussion of the basics of disc performance as they relate solely to the disc drive. The components of a disc transaction and associated performance are discussed as a means of explaining how HP calculates data sheet performance statistics.

The second half explains some of the factors that influence disc performance on HPUX. Block size is discussed as it affects performance. Disc location and its associated performance implications are discussed. And lastly, multiple unit activity is discussed as a means to increase performance. Performance measurements are used where available.

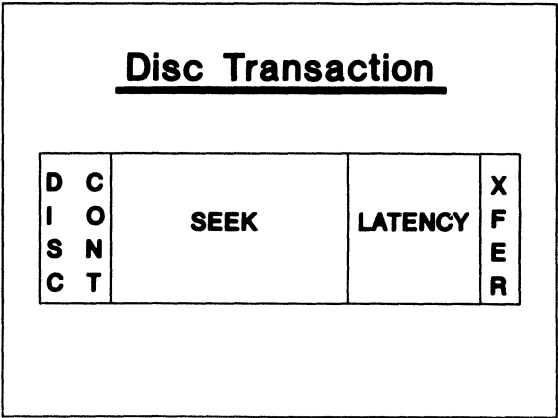
DISC BASICS

Any discussion of performance naturally begins with an understanding of how the performance is measured or calculated. The following text explains HP's definition of a disc transaction and how the performance of each of its components are measured or calculated, independent of system parameters.

* UNIX is a trademark of AT&T Bell Laboratories

Disc Transaction

A disc transaction is comprised of four components: disc controller overhead, seek time, latency and the actual transfer of information. The mechanical components, seek and latency, respectively comprise the greatest part of the transaction time. Controller overhead is the next largest component and then transfer time.



The following table compares the average transaction components of Hewlett-Packard disc drives:

Transaction Components					
	controller	seek	latency	transfer	
7912	4.0	27.1	8.3	1.2	ms
7914	4.0	28.1	8.3	1.2	ms
7933	4.5	24.0	11.1	1.0	ms
7945	10.1	30.0	8.3	2.0	ms

Disc Controller

The **disc controller** provides the intelligence of the transaction electronically. It begins processing the transaction by:

- o Decoding the disc command sent by the host computer;
- o executing the command;
- o and reporting the execution status back to the host.

Controller overhead is sacrificed for the sake of controller intelligence. However, experience has allowed HP to make controllers efficient in doing the greatest amount of work in the smallest amount of time.

Seek Time

Once the controller has decoded the command, the disc must perform some mechanical functions to prepare for command execution. The drive must first find the desired disc location by moving its heads to the desired media track. This action is defined as the **seek**.

The time to find the desired track varies with the track location and the current position of the heads. A more accurate estimate of seek time is the average seek. It is defined as the time to do all possible seeks divided by the total number of seeks possible.

Latency (or Rotational Delay)

Once the drive has found the correct track it must then find the desired sector on that track. The media continues to rotate beneath the heads as the track is searched. The time required for one rotation of the disc is defined as the **latency**. The latency, like the seek, is mechanical.

This definition of latency is certainly a "worse case" time, since the head is generally closer to the desired sector than one full rotation. The average latency is a more accurate measure and is defined as the time to complete one half a rotation.

Transfer

The **transfer** is defined as the actual movement of data from the cpu to the disc, or vice versa.

HP defines the average transfer as the average rate that data comes off the disc when reading an entire sector. Multiples of full sectors are always transferred to optimize performance.

Transaction Time

Each of the components of a disc transaction contribute to the total time involved in completing that transaction. The summation of the average time it takes to complete each component measures the average time to perform such a transaction.

The total average transaction time is defined as follows:

$$\begin{array}{r} \text{Average Controller Overhead} \\ + \text{Average Seek Time} \\ + \text{Average Latency} \\ + \text{Average Time to Transfer 1 kbyte} \\ \hline \text{Total Average Transaction Time} \end{array}$$

The following figures are calculated transactions times for Hewlett-Packard disc drives:

Transaction Times

7912	40.6 ms
7914	41.6 ms
7933	39.6 ms
7945	50.4 ms

Performance Metric

Hewlett-Packard uses the metric of I/O per second to measure disc performance. I/O per second is defined as the maximum number of disc transactions per second that a specific drive can perform at a transfer size of 1 kbyte. This measure is calculated by taking the inverse of the total transaction time that was just described. I/O per second is a measure of raw disc performance and does not take into account system specifics.

The following figures are the I/O per second calculation for HP products:

Disc Performance

7912	24.6 I/O per second
7914	24.0 I/O per second
7933	25.3 I/O per second
7945	20.0 I/O per second

PERFORMANCE FACTORS

There are several factors that influence disc performance on HPUX. The preceding discussion on disc basics does not take into account system or application specifics, but is a good measure of the relative performance as it relates to HPUX and describes some of the factors that affect performance.

Block Size

It is possible to replace the above calculated data sheet statistics with figures actually measured on HPUX. Because these measurements do not go through the file system they should be used to measure relative performance rather than absolute.

Data sheet transfer rates are calculated for transfer sizes of 1 kbyte blocks. Due to the flexibility in configuring HPUX systems, it is useful to examine transfer rates at various block sizes. For instance, transfer sizes of 8 kbyte blocks and 4 kbyte blocks are very important to Series 300 HPUX. The default file system on Series 300 HPUX attempts to transfer data in blocks of 8 kbytes and will fragment small transfers into 4 kbyte blocks.

The following data is a measure of disc throughput on Series 300 and 500 HPUX for block sizes varying from 256 bytes to 65536 bytes. The test used to obtain the data measures the time required to perform random disc reads of the varying sizes. Actual performance will vary with system and application.

Disc Performance *
Series 300 HPUX

size (bytes)	7912	7914	7945
256	6.9	6.7	5.9
512	13.6	13.2	11.8
1024	27.1	26.0	22.4
2048	52.5	51.2	42.7
4096	99.8	97.2	81.1
8192	182.6	176.0	141.5
16384	313.1	306.1	205.1
32768	475.1	469.1	261.1
65536	632.7	633.7	301.1

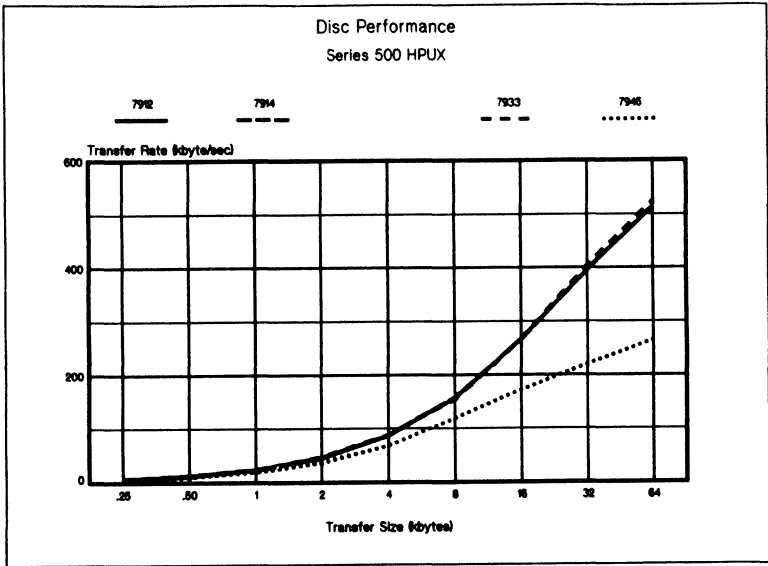
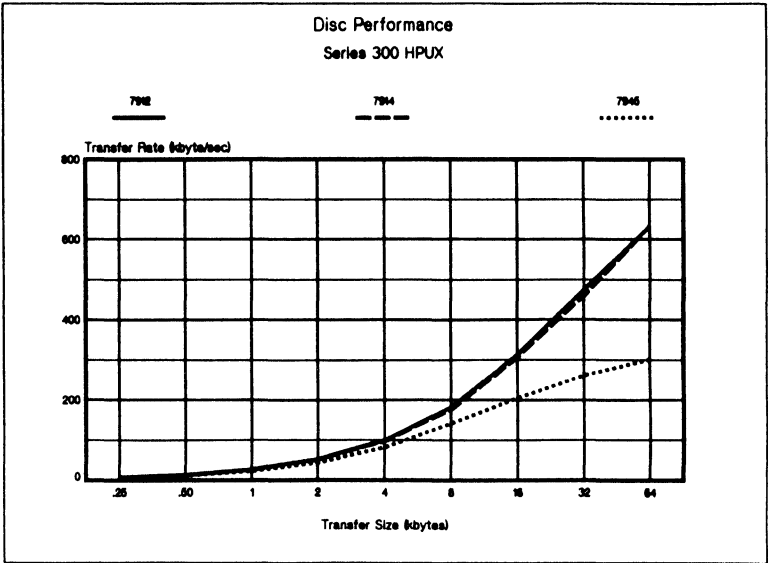
* measured in kbytes / sec

Disc Performance *
Series 500 HPUX

size (bytes)	7912	7914	7933	7945
256	5.9	5.9	6.3	5.1
512	12.3	12.0	12.4	10.0
1024	23.7	23.6	23.6	19.2
2048	46.0	45.0	47.1	36.2
4096	87.1	86.7	88.1	76.9
8192	156.2	157.4	154.2	118.2
16384	267.9	266.6	266.3	171.8
32768	396.0	395.7	404.1	220.3
65536	514.1	515.0	525.4	265.0

* measured in kbytes / sec

The same data represented in graphical form depicts the relationship between data transfer size and transfer rate.



Disc Location

The physical location of the disc drive can also affect performance. Of specific interest is the location of the virtual memory disc and the root device. Both discs are kept very busy by HPUX. The virtual memory disc is responsible for managing logical address space, physical memory utilization, and swap space. The root device is the "root" of the file system, from which all other file systems are mounted. It contains the files required for HPUX to properly run.

For single drive configurations, the virtual memory disc and the root device are one in the same device. Because these two logical discs are accessed so frequently, it makes sense, if feasible, to split them into two separate physical discs located on separated interfaces. This will allow both drives to be accessed concurrently, thus optimizing performance.

Multiple Units

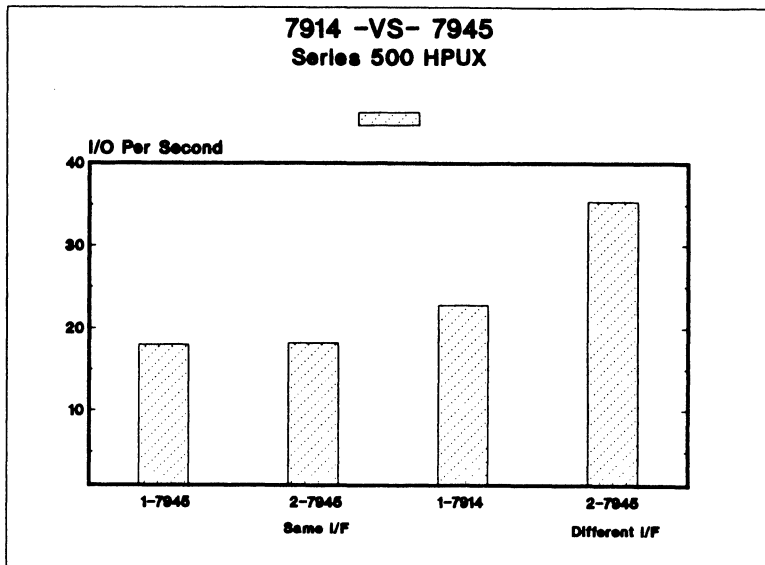
As mentioned, the interface between the disc and CPU can cause a performance bottleneck. This is often the case with HPUX because of its frequent disc accesses. There are, however, a few ways to allieviate this contention.

One way to allieviate bus contention is to spread multiple disc drives across multiple interfaces. The success of this is dependent upon the application. For instance, two drives on separate interfaces can increase I/O while running multiple processes that access both discs. The CPU can then offload to the I/O processor and keep both drives busy. Thus, the performance increase.

Two discs will not help the user running a single disc application. Remember, you only benefit if you can keep both drives busy. Two discs on the same interface will not improve performance, either. Only one device can have the bus at a time.

For example, the following graph displays results (in I/O per second) of a benchmark running two processes on Series 500 HPUX. Each process accesses a disc. There is little difference in one 7945 versus two 7945 drives on the same

interface (18.2 vs. 18.0). The 7914 is a higher performer (22.8) and two 7945 drives on separate interfaces (35.4) shows the best performance.



Another way to reduce bus contention is to place the root disc and cartridge tape on separate interfaces. In this configuration, both the tape and disc can be accessed simultaneously. Also, disc accesses can occur while the tape is loading and unloading.

These are just a few of the ways interface contention can be relieved. Systems with multiple users and multiple processes should consider either of the above suggestions, or both.

SUMMARY

The disc transaction is comprised of many components. Each component, in turn, impacts overall disc performance. These performance factors are fixed, and cannot be tuned by

the HPUX user. The understanding of these static factors provides insight into the operation of the disc drive and, ultimately, overall disc performance.

There are several factors affecting disc performance on HPUX that the user does have control over. By examining disc performance at the various transfer lengths, rather than at the fixed length of 1 kbyte, the user can select the mass storage that best suits the way in which the system is used. Location of the mass storage should also be examined for optimal performance. In addition, for those systems where frequent multiple processing is going on, performance may be improved with multiple disc/tapes on separate interfaces.