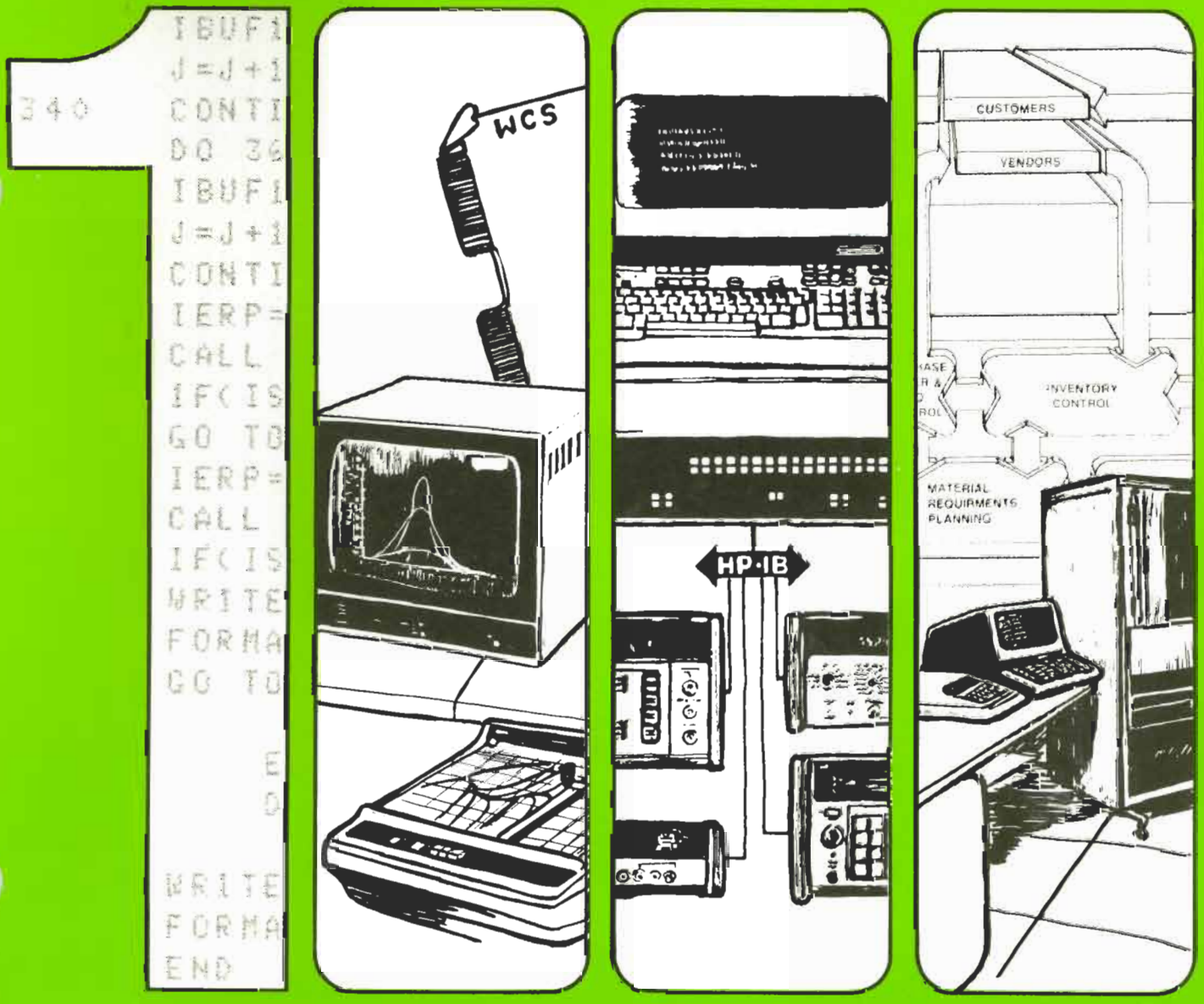
HEWLETT PACKARD

Hewlett-Packard Computer Systems

COMMUNICATOR



1980

Volume IV Issue 4

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HEWLETT-PACKARD
COMPUTER SYSTEMS

Volume IV
Issue 4

COMMUNICATOR/1000

Feature Articles

Departments

EDITOR'S DESK

ABOUT THIS ISSUE

This issue of the Communicator/1000 features two articles in section OPERATING SYSTEMS and one each in OPERATIONS MANAGEMENT and LANGUAGES. All the articles were written by HP employees; two come from the field and two come from Data Systems Division.

In the OPERATING SYSTEMS section Alan Housley and Clark Johnson of DSD's technical marketing department explain the RTE-IVB version of their CLEXT program for clearing extents. The original version, "Creating and Clearing Extents", appeared in the Communicator/1000, Volume III, Issue 1. The second article in this section is the fifth article in the LINKS/1000 series. The articles in this series are part of an unpublished manual contributed by Bob Niland of HP's Lexington sales and service office.

In the LANGUAGES section John Pezzano of HP's El Paso sales and service office wrote a fine article on how to make your FORTRAN code execute more quickly.

Finally, in the OPERATIONS MANAGEMENT section, Carol Jonas of Data Systems Division wrote an article explaining the need for a BASIC/IMAGE interface (BAIMG) and how the interface is done.

The winners of the calculators for this issue are:

Best Feature Article by an HP Division Employee not in Data Systems Technical Marketing	FUNCTIONS OF A BAIMG Carol Jonas
Best Feature Article by an HP Field Employee	FAST FORTRAN John Pezzano

I hope you enjoy this issue of the Communicator/1000!

LETTER TO THE EDITOR

Dear Sir,

I am having trouble using the FMGR :MC command with my user- written floppy disc driver (for AED 2500). What does the MC command look for before it allows a cartridge to be mounted? Does it check for driver numbers (30,31,32,33, etc.), track map table, whatever? For RTE II,III, and IVA I finally got my floppy mounted by naming the driver DVR30 and putting the number of tracks in the :MC command. After generating RTE-IVB the command :MC,-40,63 returns FMGR-001. Control commands to the driver (formatting, for example) work fine.

Sincerely,

Michael M. Thomas

Dear Mr. Thomas,

The :MC command does check driver numbers (i.e. type) to make sure that the device is a disc so you were correct to change the name of your driver. The FMGR-001 is being returned because your driver rejected a special EXEC call issued by the :MC command. This EXEC call is documented on page 2-3 of the DVR32 manual (92068-90012). An EXEC read in which the track number (ITRAK) is greater than the size of the subchannel, or in which the track number = −1, should cause the driver to return in the B register the number of tracks on the subchannel, and set bit 5 in EQT word 5 to 1. The driver should also return the number of 64-word sectors per track in IBUFR(1).

Sincerely,

The Editor

EDITOR'S DESK

BECOME A PUBLISHED AUTHOR IN THE COMMUNICATOR/1000 . . .

The COMMUNICATOR is a technical publication designed for HP 1000 computer users. Through technical articles, the direct answering of customers' technical questions, cataloging of contributed user programs, and publication of new product announcements and product training schedules, the COMMUNICATOR strives to help each reader utililize their HP 1000's more effectively.

The Feature Articles are clearly the most important part of the COMMUNICATOR. Feature Articles are intended to promote a significant cross-fertilization of ideas, to provide in-depth technical descriptions of application programs that could be useful to a wide range of users, and to increase user understanding of the most sophisticated capabilities designed into HP software. You might think of the COMMUNICATOR as a publication which can extend your awareness of HP 1000's to include that of thousands of users worldwide as well as that of many HP engineers in Data Systems factories at Cupertino, California and Grenoble, France.

To accomplish these goals, editors of the COMMUNICATOR actively seek technical articles from HP 1000 customers, HP Systems Engineers in the Field, and Marketing and R&D Engineers in the factories. Technical articles from customers are most highly valued because it is customers who are closest to real-world applications.

WIN AN HP-32E CALCULATOR!

Authoring a published article provides a uniquely satisfying and visible feeling of accomplishment. To provide a more tangible benefit, however, HP gives away three free HP-32E hand-held calculators to Feature Article authors in each COMMUNICATOR/1000 issue! Authors are divided into three categories. A calculator is awarded to the author of the best Feature Article in each of the author categories. The three author categories are:

1. HP 1000 Customers;

2. HP field employees;

3. HP division employees not in the Data Systems Division Technical Marketing Dept.

Each author category is judged separately. A calculator prize will be awarded even if there is only one entry in an author category.

Feature Articles are judged on the following bases: (1) quality of technical content; (2) level of interest to a wide spectrum of COMMUNICATOR/1000 readers; (3) thoroughness with which subject is covered; and, (4) clarity of presentation.

What is a Feature Article? A Feature Article meets the following criteria:

1. Its topic is of general technical interest to COMMUNICATOR/1000 readers;

2. The topic falls into one of the following categories —

   OPERATING SYSTEMS
   DATA COMMUNICATIONS
   INSTRUMENTATION
   COMPUTATION
   OPERATIONS MANAGEMENT

3. The article covers at least two pages of the COMMUNICATOR/1000, exclusive of listings and illustrations (i.e., at least 1650 words).

There is a little fine print with regard to eligibility for receiving a calculator; it follows. No individual author will be awarded more than one calculator in a calendar year. In the case of multiple authors, the calculator will be awarded to the first listed author of the winning article. An article which is part of a series will compete on its own merits with other articles in the issue. The total of all articles in the series will not compete against the total of all articles in another series. Employees of Technical Marketing at HP's Data Systems Division factory in Cupertino are not eligible to win a calculator.

All winners of calculators will be announced in the issue of the COMMUNICATOR/1000 in which their articles appear. Again, all Feature Articles are judged by an impartial panel of three DSD Technical Marketing Engineers.

A SPECIAL DEAL IN THE OEM CORNER

When an HP 1000 OEM writes a Feature Article that is not only technically detailed and insightful but also application-oriented as opposed to theoretical, then that OEM may ask that the article be included in THE OEM CORNER. A Feature Article included in THE OEM CORNER may contain up to 150 words of pure product description as well as a picture or illustration of the OEM'S product or its unique contribution. HP's objective is twofold: (1) to promote awareness of the capabilities HP 1000 OEMs' products among all HP 1000 users; and, (2) to publish an article of technical interest and depth.

IF YOU'RE PRESSED FOR TIME . . .

If you are short of time, but still have that urge to express yourself technically, don't forget the COMMUNICATOR/1000 BIT BUCKET. It's the perfect place for a short description of a routine you've written or an insight you've had.

THE MECHANICS OF SUBMITTING AN ARTICLE

If at all possible please submit an RTE File containing the text of your article recorded on a Minicartridge (preferrably) or on a paper tape along with the line printer or typed copy of your article. This will help all of us to be more efficient. The Minicartridge will be returned to you promptly. Please include your address and phone number along with your article.

All articles are subject to editorship and minor revisions. The author will be contacted if there is any question of changing the information content. Articles requiring a major revision will be returned to the author with an explanatory note and suggestions for change. We hope not to return any articles at all; if we do, we would like to work closely with the author to improve the article. HP does, however, reserve the right to reject articles that are not technical or that are not of general interest to COMMUNICATOR/1000 readers.

Please submit your COMMUNICATOR/1000 article to the following address:

Editor, COMMUNICATOR/1000
Data Systems Division
Hewlett-Packard Company
11000 Wolfe Road
Cupertino, California 95014
USA

The Editor looks forward to an exciting year of articles in the COMMUNICATOR/1000.

With best regards,

The Editor

EASY WAY TO MOUNT/DISMOUNT ALL GROUP/PRIVATE CARTRIDGES

Ian Higgins/HP Winnersh, U.K

Have you ever regenerated a system and had subsequently to log-on as every user in order to re-mount their private and group cartridges? It's very tedious! A much easier way is to use a procedure file and submit it to JOB as follows:

```
:RU,JOB,MCUSER
```

where MCUSER is:

```
:JO,USERS,USER.GROUP/PASSWORD
:MC,LU1,P
:MC,LU2,G
:EO
```

You can have mount cartridge procedure files for every user on the system and thereby mount all group/private cartridges without ever logging on or off.

LOADING GRAPHICS ID's DYNAMICALLY

Vladimer Preysman/HP Santa Clara

Are you having problems loading graphics programs? Are you getting memory overflow because of device subroutines which are appended to your program even if you don't need them? Would you like to assign graphics IDs dynamically without editing and reassembling &DLTBL?

If YES here is an idea...

In a program:

```
FTN4,Q
      -----
C
C     SET UP FIRST ID (LET IT BE 2648)
C
      CALL A2648
C
C     THIS CALL WILL PUT ENTRIES TO 2648'S DEVICE SUBROUTINES (DVG01 & DCT01)
C     INTO DEVICE LINK TABLE
C
C     SET UP SECOND ID (FOR EXAMPLE 7221)
C
      CALL A7221
      -----
C
C     AND SO ON...
C
C     ENTRY POINTS AVAILABLE: A2648
C                             A2608
C                             A7221, B7221, S7221
C                             A7225
C                             A7245
C                             A9872, B9872, S9872
C                             A9874
C
      -----
C
C     NOW DO ALL GRAPHICS CALLS USING ID=1 FOR 2648, ID=2 FOR 7221
C
```

When loading your program after relocating the program itself search file %DNTBL which contains the dynamic ID table. This file consists of

A2648
A2608
A7221, B7221, S7221
A7225
A7245
A9872, B9872, S9872
A9874
DLTBL SPACE TO BUILD A TABLE

LOADR will pick only the ones that are called (all of them have DLTBL declared as EXT). If you need to add entry points for other devices just follow the examples below and append relocatables in front of DLTBL (use MERGE in RTE-IVB).

```
0001                    ASMB,Q              *** DYNAMIC DEVICE LINK TABLE ***
0003  00000                   NAM DLTBL
0004                          ENT DPTR,ADGRD
0005                          EXT EXEC
0006*
0007  00000           A       EQU 0
0008  00000 000000R   PNTR    DEF TABLE-2   LAST LINK POINTER
0009  00001 000000    DPTR    DEC 0         NUMBER OF DEFINED ENTRIES
0010                          SUP
0011                    TABLE REP 10        LET IT BE 10 DEVICES
0012  00002 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00004 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00006 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00010 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00012 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00014 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00016 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00020 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00022 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0012  00024 000000            OCT 0,0       SPACE FOR 2 LINKS (DVTXX, DCTXX)
0013                          UNS
0014  00026 000026R ENDTB DEF *             END-OF-TABLE POINTER
0015*
0016  00027 000000    ADGRD NOP
0017  00030 000000R           LDA PNTR      GET LAST LINK POINTER
0018  00031 002004            INA            ADVANCE IT
0019  00032 002004            INA              TWICE.
0020  00033 000026R           CPA ENDTB     IS TABLE FILLED?
0021  00034 000051R           JMP OVFL       -YES PROCESS OVERFLOW
0022  00035 000000R           STA PNTR       -NO STORE NEW POINTER
0023  00036 000001R           ISZ DPTR      INCREMENT LINK
0024  00037 000001R           ISZ DPTR       COUNTER TWICE.
0025  00040 104200            DLD ADGRD,I   GET THE LINKS
0026  00042 104400            DST PNTR,I     AND PUT THEM INTO TABLE.
0027  00044 000027R RETN      LDA ADGRD     GET 1-ST LEVEL
0028  00045 000073R           ADA =B-2       RETURN ADDRESS
0029  00046 000000            LDA A,I          FIND OUT
0030  00047 000000            LDA A,I            INITIAL RETURN
0031  00050 000000            JMP A,I       AND GET OUT.
0032*
```

```
0033  00051 000001X OVFL  JSB EXEC
0034  00052 000057R       DEF END
0035  00053 000060R       DEF WRITE
0036  00054 000061R       DEF LU.1
0037  00055 000063R       DEF OVER
0038  00056 000062R       DEF OVER-1
0039  00057 000044R END    JMP RETN
0040*
0041  00060 000002  WRITE OCT 2
0042  00061 000001  LU.1  DEC 1
0043  00062 000010        DEC 8
0044  00063 042114  OVER  ASC 8,DLTBL OVERFLOW!
0045                      END
**  NO ERRORS *TOTAL **RTE ASMB 92067-16011**
```

```
0001                ASMB,Q
0002  00000               NAM A2648     DEVICE LINK FOR 2648 12-14-79VP
0003                      ENT A2648
0004                      EXT DCT01,DVG01,ADGRD
0005  00000 000000  A2648 NOP
0006  00001 000003X       JSB ADGRD
0007  00002 000002X       DEF DVG01
0008  00003 000001X       DEF DCT01
0009*       ...AND WE'LL NEVER COME BACK
0010                      END
**  NO ERRORS *TOTAL **RTE ASMB 92067-16011**
```

```
0001                ASMB,Q
0002  00000               NAM A7221     DEVICE LINK FOR 7221 12-14-79VP
0003                      ENT A7221,B7221,S7221
0004                      EXT DCT05,DVG05,ADGRD
0005  00000 000000  A7221 NOP
0006  00001 000003X       JSB ADGRD
0007  00002 000002X       DEF DVG05
0008  00003 000001X       DEF DCT05
0009*       ...AND WE'LL NEVER COME BACK
0010  00000         B7221 EQU A7221
0011  00000         S7221 EQU A7221
0012                      END
**  NO ERRORS *TOTAL **RTE ASMB 92067-16011**
```

THE SST IN THE SESSION CONTROL BLOCK

Martha Slettedahl/HP Data Systems Division

I get a great number of questions regarding LGON errors, and after doing some digging in the System Manager's manual as well as talking with various DSD lab engineers I have compiled a simple explanation of how the Session Control Block fits into the logon process. In addition, I will explain how the SL, MC, AC and DC commands are recorded.

The maximum size of the SST in the session control block is determined as follows. The disc limit (word 31 of the user account entry) is added to the number of SST spares (lower 8 bits of word 32 of the user account entry). (These values are set when the user's account is defined with the ACCTS program and can be seen with the LI command in ACCTS). To this total is added the number of system LU/session LU mappings defined for the user's account and station, plus the number of system disc LU's.

MAX SIZE = disc limit + #SST spares + station & user LU's + system discs

When a user logs on to the system the SST in the Session Control Block is built by the program LGON. First, the LU's for the logon station are obtained from the configuration table (pg. J-5 in the System Manager's manual). Next, the LU's for that user are obtained from the user and group account entries (see pgs. J-10 and J-9 in the System Manager's manual). (The mappings obtained from the user and group accounts override those obtained from the configuration table so that duplicate entries are not added.) Finally, the cartridge directory is checked for system disc LU's. When a user tries to log on, if the MAX SIZE of his SST is greater than 63 entries a LGON 09 error will occur. Note that the ACCTS program cannot check for this error because the MAX SIZE is partially determined by the number of station LU's (different for each terminal).

In addition to the SST, the session control block contains a table which keeps track of the number of mounted disc cartridges. I will call this table the disc cartridge table. The size of the disc cartridge table is equal to the disc limit, and entries are filled at logon for each private and group cartridge in the user's account (these discs are mounted by LGON). If the number of private and group cartridges exceeds the size of the table (disc limit) a LGON 11 will occur.

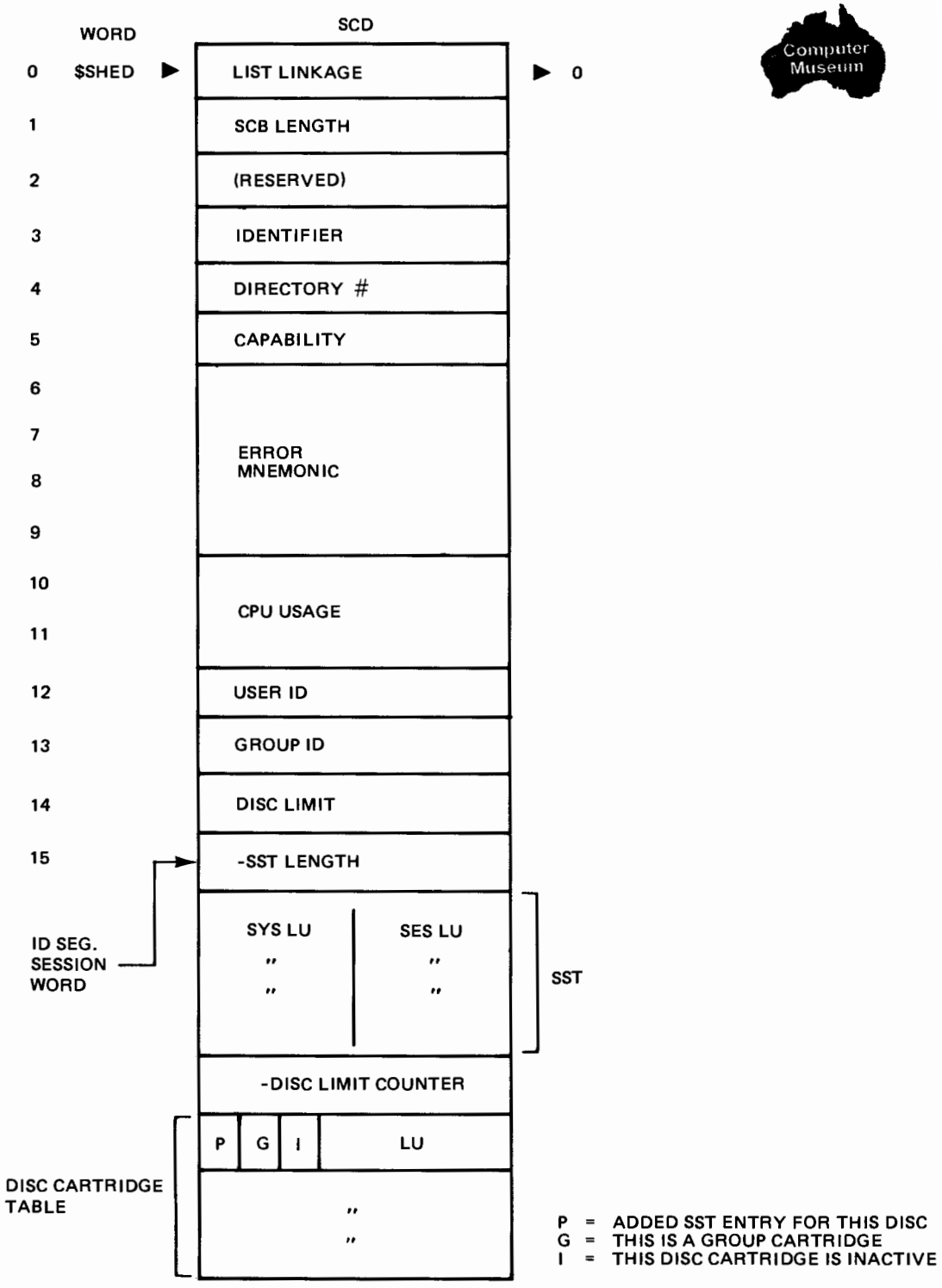
Before logon the number of free entries in the SST equals the disc limit plus the #SST spares (these are defined in ACCTS). At logon some of these entries are used for private and group disc LU's. After logon, then, the number of free entries is reduced by the number of private and group cartridges. The term given to these free entries is SST spares. Remember that this is not the same as the "#SST spares" term used in ACCTS. Therefore, immediately after logon we have:

#actual SST spares = disc limit + #SST spares - #private & group discs

Actual SST spares are used whenever the SL command is used to add an LU mapping to the SST. A spare entry is also used by the AC command. The MC command does not add an entry to the SST. Instead, the LU for the cartridge to be mounted must be added to the SST with the SL command before the MC command is invoked. The SL command can be used to remove entries from the SST which were added with an SL (:SL,lu,–). In addition, these entries are removed at logoff.

Each MC and AC command is recorded in the disc cartridge table. Remember that the size of this table is equal to the disc limit so no more cartridges can be mounted when that limit has been reached. The DC command is used to remove entries from the disc cartridge table.

SESSION CONTROL BLOCK



HOW TO DIRECT FORMATTER ERROR MESSAGES TO ANY LU

Gary Ericson/HP Data Systems Division

Occasionally the question arises, "How do I keep the Fortran Formatter errors from going to the line printer?" The answer is in the DOS/RTE Relocatable Library Reference Manual, but it sometimes gets overlooked.

The entry point FMT.E is "hard-coded" to the value 6, and the Formatter uses this value as the LU for its error messages. Creating a dummy entry point with a different value, and loading this entry point with your program, will change where the messages are sent. Typically this would be set to "1" like this:

```
ASMB,L
       NAM FMT.E
       ENT FMT.E
FMT.E  DEC 1
       END
```

Going one step further, you could create a Fortran-callable subroutine that could be used to set the entry point FMT.E at run time. This might be done like this:

```
ASMB,L
       NAM FMTE
       ENT FMTE,FMT.E
       EXT .ENTR
LU     DEF FMT.E        PARAMETER DEFAULTED TO CURRENT FMT.E VALUE
*
FMTE   NOP
       JSB .ENTR        GET PARAMETER PASSED IN, IF ANY
       DEF LU
       LDA LU,I         PICK UP LU#,
       STA FMT.E        PUT INTO ENTRY POINT FMT.E,
EXIT   JMP FMTE,I       AND RETURN
*
FMT.E  DEC 1            DEFAULT LU TO 1
*
       END
```

This calling sequence for this is:

```
     CALL FMTE
or   CALL FMTE(LU)
```

where LU is the device where the error messages will be sent

Loading this subroutine with your program will default FMT.E to 1, and calling it from a program can change the LU to anything you want. This same procedure can be used for other "hard-coded" entry points such as ER0.E (the error LU for the relocatable library).

MORE WITH CLEXT: Clearing Extents On An RTE-IVB System

Alan K. Housley and Clark Johnson/HP Data Systems Division

Appearing in a previous Communicator, Volume III, Issue 1 is an article, Creating and Clearing Extents, describing the process of file extent creation and clearing for RTE-IVA operating systems. Since the release of RTE-IVB, I have had numerous requests to update the extent clearing routine (CLEXT) that was included in the article so that it will operate under RTE-IVB.

Many of our new readers may not have access to the article to which I am referring. Therefore, for their benefit, I will repeat much of the information about extent creation in the following pararaphs. Readers familiar with the article may want to skip ahead to subheading REUSE OF DISC SPACE. However, there have been a few changes since the last article so you may want to read on for review and to obtain new information.

EXTENT CREATION (Review)

Anyone who has used the file manager DL command any number of times has surely noticed that some of the file names are followed by a three digit number to the right of the file's block size. This number signifies that the file entry is an extent of a main file by the same name. For example:

```
NAME    TYPE   BLKS/LU OPEN TO
          .
          .
EXAMPL 00004   00048
EXAMPL 00004   00048   +001    <---- File extension #1
OUTPT  00004   00020                 for file EXAMPL
&NEEDS 00004   00010
OUTPT  00004   00020   +001    <---- File extension #1
OUTPT  00004   00020   +002    <---- File extension #2
SYSMGR 00004   00001                 for file OUTPT
CODES  00004   00005
FILES  00003   00017
SYSMGR 00004   00001   +001
%TIGER 00005   00019
```

An extent (short for file extension) is automatically created by FMP with the same name and size as the original file (files type 3 and above) when a write request points to a location beyond the range of the currently defined file. Each extent is identified by an extent number that is stored in the upper byte of word five of the corresponding entry in the file directory. Whenever another extent is created by FMP, the extent number is incremented and stored in the file directory entry as described above.

EXAMPLE: Shown in the following figure is the file directory entry for the file EXAMPL:11156:20:3:1, and its first file extension.

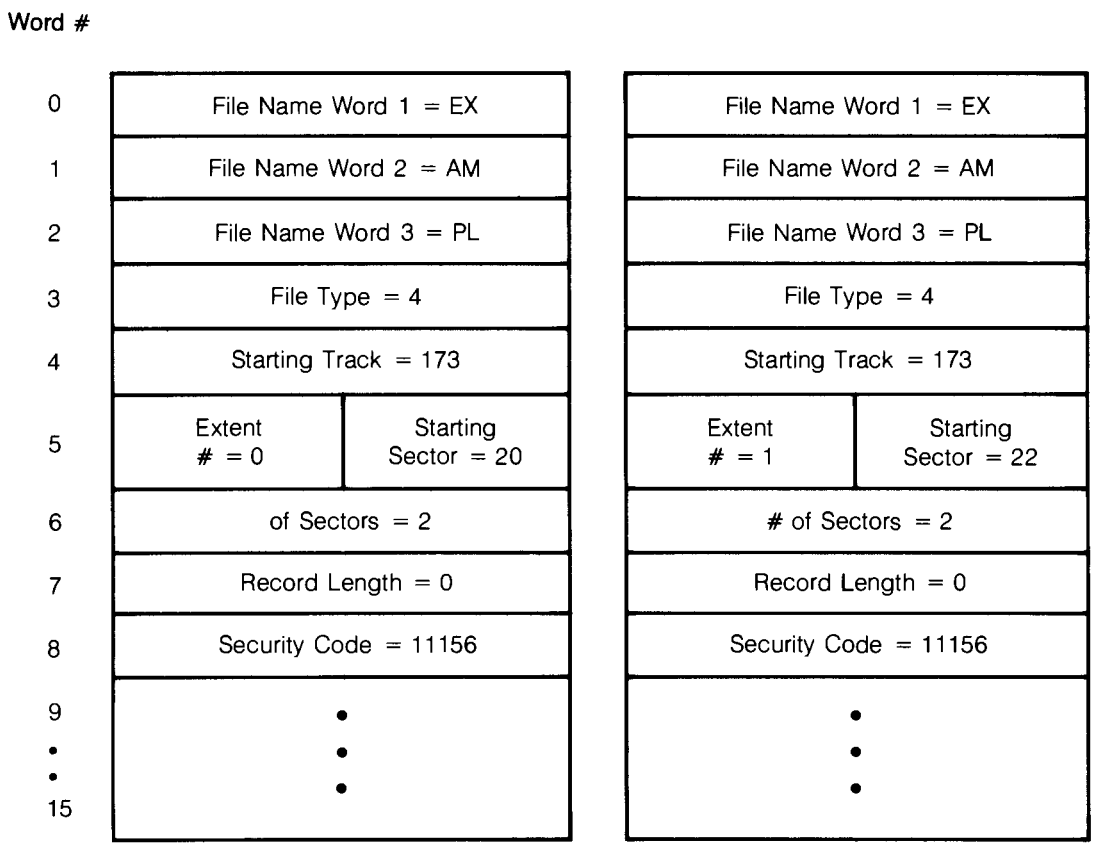


Figure 1

FMP will repeat the process of creating extents as many times as necessary as a file increases in size. Only a full disc LU (FMGR -033), a full file directory (FMGR -014), or an extent number exceeding 255 (FMGR -046) will abort the process.

Everytime a file or part of a file that contains extents is accessed, the file directory entry for each extent must be read in order to find the location of the file extension on the disc. This of course slows down the file access process. Obviously, to speed up the process, all file extensions might be combined into one file and, therefore, only one entry from the file directory would have to be read to locate the entire file.

EXTENT CLEANUP

To combine file extensions into one continuous file is a simple process of storing the file into a temporary file, purging the original file, and renaming the temporary file to the original file name. As yet, there is not a supported routine that clears up file extensions in this manner, so many people have written transfer files that clear extents on their disc LU's. One version of such a routine is:

```
:ST,Original File Name:SC:CR#,Temp. File Name:SC:CR#::-1,SA
:PU,Original File Name:SC:CR#
:RN,Temp. File Name:SC:CR#,Original File Name
```

Put in the form of a transfer file using globals, this routine will process one file at a time. However, when several files need to be cleared of extents, the user must obtain all the file names by using the DL command and enter each file name into the transfer file one at a time. I personally find this process tedious, and consequently wrote a program to find all files with extents for a particular disc LU and clear them. I can simply run the program from time to time to keep my disc LU "cleaned up", and I don't need to know which files have gained extents since the last "clean up".

As mentioned earlier, the main advantage of CLEXT is to cause fewer disc accesses than would normally occur when reading each directory entry for a particular file and its extents. Also, CLEXT gives many people that "Warm Fuzzy Feeling" when they don't see any file extensions scattered on their discs when viewing their cartridge directories.

This whole discussion is fine for those people who need fast disc access, and have plenty of disc space (or don't mind packing their discs from time to time). The discussion that follows will show another side to the disc usage story, and will also show reasons for a new feature added to CLEXT.

REUSE OF DISC SPACE

When a disc file is purged by FMP, a "logical hole" is left on the disc where the file once resided. That hole can be filled in with a new file if the new file is the same size as the original file that was purged. FMP determines if a file has been purged, and therefore empty disc space exists, by reading the first word of the 16 word file directory entry. If there is a negative one (−1) in the first word of the entry, the file has been purged. FMP can then determine if the "hole" left by the purged file is the right size to accomodate a new file or file extent by reading word 6 of the file directory entry.

EXAMPLE: Suppose a file called SUMMIT and its extent, 48 blocks long total, starting at track 168 and sector 18. The resulting file directory entries are:

Word #		
0	File Name Word 1 = SU	File Name Word 1 = SU
1	File Name Word 2 = MM	File Name Word 2 = MM
2	File Name Word 3 = IT	File Name Word 3 = IT
3	File Type = 3	File Type = 3
4	Starting Track = 168	Starting Track = 168
5	Extent # = 0 / Starting Sector = 18	Extent # = 1 / Starting Sector = 66
6	# of Sectors = 48	# of Sectors = 48
7	Record Length = 0	Record Length = 0
8	Security Code = 0	Security Code = 0
9 • • • 15	• • •	• • •

Figure 2

When SUMMIT is purged, word 0 of each file directory above will be set to negative one (−1).

Now, a new file, %PROG:11111:LU, 24 blocks long, can reside in the same location as SUMMIT's main file. The file directory entry for the first extent of SUMMIT is left unchanged since %PROG only required 24 blocks. The resulting file directory entries are:

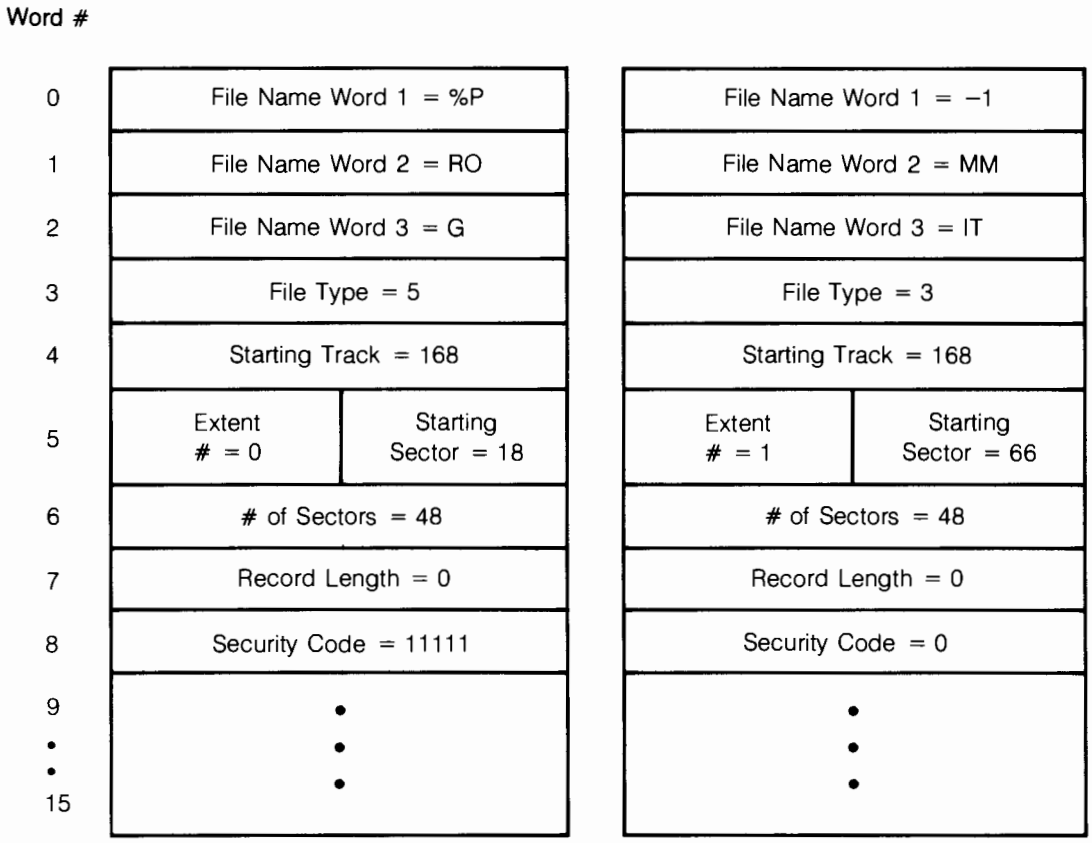


Figure 3

What I intend to show by this example is that disc space left by purged files can be reused by files or extents of equal size. If the file is purged and all newly created files or extents are larger than the purged file, the original files disc space will never be reused. The only way to recover this unused area is to pack the disc.

Therefore, a good practice, in order to obtain very efficient disc usage, is to make all files and their extents the same size. For every purged file and its extents, a new file will be able to reuse the unoccupied space left because of the purge. Those who use their disc space heavily will find by following this practice, their discs will become filled less often. This is especially helpful for someone in a session environment who doesn't have the capability to pack his disc.

NEW CLEXT FEATURE

In light of the above information, I have added a feature to CLEXT that allows the user to specify the size of the main file and all extents. The capability to clear extents will still exist if desired.

Therefore, CLEXT now has two features to satisfy two groups of people:

1. CLEXT will allow you to clear all extents to gain faster disc access, and

2. CLEXT will allow you to set all files with extents to a specified size.

A note to those who will be using the new feature of CLEXT:

It will take a little time to get your disc to the state of most efficient use. That is, after CLEXT has been run a few times, your disc may become filled up and require packing. This is because there were possibly a lot of small files with small extent sizes which were changed to the new size you specified. These small disc areas will therefore be left unused as we discussed earlier in the article. Once you pack the disc, all the small areas will be recovered and your disc space will start being utilized more efficiently. Also, I recommend that you choose a size of 24 blocks for files and their extents when running CLEXT, or when creating files. A block size of 24 has been found to work very effectively.

PROGRAM NOTES

To invoke CLEXT:

```
:RU, CLEXT [,list lu [,disc lu[,file and extend size]]]
```

where: list lu = list device (default is user console)
disc lu = disc logical unit to process extents
file and extent size = integer value specifying the block size of extents
−1 = clear extents
0 = abort

If no parameters are provided in the runstring, CLEXT will prompt for the disc lu and extent size, but will default the list lu to the user terminal.

If LU 2 or LU 3 is entered as the disc lu number, CLEXT will prompt for another LU number since these disc LU's are considered system disc areas for RTE-IV systems. If for some reason CLEXT is unable to process a file, the program will issue an FMP error describing why the file could not be processed and the program will terminate. Your file will be left intact if such a situation occurs.

```
 PAGE 0001  FTN.   8:42 AM  TUE., 20  MAY , 1980


0001  FTN4,L
0002        PROGRAM CLEXT(),AKH'S AND CJ'S CLEAR EXTENTS
0003
0004  C*********************************************************************
0005  C                                                                    *
0006  C  NAME:      CLEXT (CLEAR EXTENTS)                                 *
0007  C                                                                    *
0008  C  DATE:      1/11/79                                               *
0009  C                                                                    *
0010  C  REVISED:   12/14/79                                              *
0011  C              3/27/80                                               *
0012  C                                                                    *
0013  C  VERSION:   AKH.03                                                *
0014  C                                                                    *
0015  C  AUTHOR:    ALAN K. HOUSLEY                                       *
0016  C             DSD TECHNICAL MARKETING                               *
0017  C                                                                    *
0018  C  PURPOSE:   ROUTINE TO EITHER CLEAR ALL EXTENTS ON A SPECIFIED DISC  *
0019  C             LU OR TO PROCESS ALL FILES WITH EXTENTS TO A SPECIFIED  *
0020  C             SIZE (SUCH AS 24 BLOCKS).                              *
0021  C                                                                    *
0022  C  INVOKE:    RU, CLEXT, LIST LU, DISC LU, FILE AND EXTENT SIZE      *
0023  C                                                                    *
0024  C*********************************************************************
0025
0026
0027        IMPLICIT INTEGER (A-Z)
0028
0029        INTEGER BUFFIN (10),  BUFOUT (33),
0030       *         CRNTFL (3),   DIRBUF (16),
0031       *         FILDIR(128),  STATUS(125)
0032
0033  C  OBTAIN THE USER'S TERMINAL LU
0034
0035        LLU = LOGLU(I)
0036
0037  C  OBTAIN THE LIST LU, DISC LU, AND EXTENT SIZE
0038  C  PARSE THE INPUT STRING
0039
0040        CALL GETST (BUFFIN,-20,TRNLOG)
0041        CALL PARSE (BUFFIN,TRNLOG,BUFOUT)
0042
0043
0044  C  GET THE LIST LU
0045
0046        JR = 1
0047        LISTLU = BUFOUT (JR+1)
0048        IF (BUFOUT(JR).EQ.1) GO TO 100
0049        LISTLU = LOGLU (I)
0050
0051  C  GET THE DISC LU OR THE CARTRIDGE REF # TO BE PROCESSED
0052
0053   100  JR = JR+4
0054        IF (BUFOUT(JR).GT.0) GO TO 140
0055
```

```
 PAGE 0002  CLEXT  8:42 AM  TUE., 20  MAY , 1980


0056   110  WRITE(LLU,120)
0057   120  FORMAT(" DISC LU (0 = EXIT):  ")
0058
0059        READ(LLU,*)DLU
0060   130  IF(DLU.EQ.0) GO TO 9999
0061        IF(DLU.EQ.2.OR.DLU.EQ.3.OR.DLU.GT.255) GO TO 110
0062        IF(DLU.LT.0) DLU = -(DLU)
0063        CRNUM = 0
0064        GO TO 160
0065
0066   140  IF (BUFOUT(JR).EQ.2) GO TO 150
0067        DLU = BUFOUT(JR+1)
0068        CRNUM = 0
0069        GO TO 160
0070
0071   150  CRNUM = BUFOUT (JR+1)
0072        DLU = 0
0073
0074  C  GET THE EXTENT SIZE DESIRED FOR EACH FILE
0075
0076   160  JR = JR+4
0077        SIZE = BUFOUT(JR+1)
0078        IF(BUFOUT(JR).EQ.1) GO TO 180
0079
0080  C  DETERMINE IF THE USER WOULD LIKE TO CLEAR ALL THE EXTENTS OR TO
0081  C  SET TO A SPECIFIED SIZE.
0082
0083        WRITE (LLU,170)
0084   170  FORMAT (" ENTER DESIRED SIZE OF FILE AND IT'S EXTENTS"/
0085       *         " NEGATIVE ONE (-1) WILL CLEAR ALL EXTENTS:     ")
0086        READ (LLU,*) SIZE
0087        IF(SIZE.GT.0) GO TO 180
0088        IF(SIZE.NE.-1) GO TO 9999
0089
0090  C  GET THE CARTRIDGE DIRECTORY AND DETERMINE IF THE DISC LU IS
0091  C  MOUNTED.  IF SO, GET THE CARTRIDGE NUMBER AND THE FILE DIRECTORY
0092  C  TRACK FROM THE CARTRIDGE DIRECTORY ENTRY CORRESPONDING TO THE
0093  C  DISC LU.
0094
0095   180  CALL FSTAT(STATUS)
0096        J = 1
0097        DO 210 I = 1,31
0098        IF(STATUS(J).NE.0) GO TO 200
0099        WRITE(LLU,190)
0100   190  FORMAT(" DISC NOT MOUNTED"/,)
0101        GO TO 110
0102
0103   200  IF(STATUS(J).NE.DLU.AND.STATUS(J+2).NE.CRNUM) GO TO 210
0104        DLU = STATUS (J)
0105        J = J+1
0106        DRTRK1 = STATUS(J)
0107        J = J+1
0108        CRNUM = STATUS(J)
0109
0110        GO TO 220
```

```
 PAGE 0003  CLEXT  8:42 AM  TUE., 20  MAY , 1980


0111    210  J = I*4+1
0112
0113   C  READ THE FILE DIRECTORY FROM THE USERS DISC CARTRIDGE.
0114
0115    220  CALL EXEC(1,DLU,DIRBUF,16,DRTRK1,0)
0116
0117   C  OBTAIN THE # OF DIRECTORY TRACKS, AND THE # OF SECTORS/TRACK.
0118
0119         NMDTRK = DIRBUF(9) *(-1)
0120         SECTRK = DIRBUF(7)
0121
0122
0123   C  READ 2 SECTORS AT A TIME.
0124   C  FIND THE NEXT 2 SECTORS TO READ BY USING THE FORMULA:
0125   C  SECTOR ADDRESS = (BLOCK*14) MODULO (#SECTORS/TRACK)
0126
0127   C  TKCNTR = THE DIRECTORY TRACK COUNTER.
0128   C  COUNTER = THE SECTOR COUNTER.
0129
0130
0131         TKCNTR = 0
0132
0133
0134    230  COUNTR = 0
0135    240  TEST = COUNTR*14
0136    250  IF(TEST.LT.SECTRK) GO TO 260
0137         TEST = TEST-SECTRK
0138         GO TO 250
0139    260  CONTINUE
0140
0141
0142   C  READ THE 16 WORD FILE DIRECTORY ENTRY
0143
0144   C  FIRST  DETERMINE IF  TRYING TO  PROCESS THE  DIRECTORY TRACK
0145   C  IF SO, SKIP TO NEXT FILE DIRECTORY ENTRY. IF A FILE CONTAINS
0146   C  AN  EXTENT,   IT FIRST MUST HAVE A VALUE  .GE.  ZERO IN  THE
0147   C  FIRST  WORD OF  THE FILE DIRECTORY ENTRY, AND THEN MUST HAVE
0148   C  AT LEAST 1 BIT SET IN  THE UPPER  BYTE OF  WORD 5.   IF BOTH
0149   C  ARE TRUE THEN CLEAR THE FILES EXTENTS.  IF, HOWEVER, EXTENTS
0150   C  ARE NOT TO BE CLEARED, BUT ARE TO BE SET TO A SPECIFIED SIZE,
0151   C  THEN CHECK  THE SIZE OF  THE FILE  AND ITS EXTENTS IN WORD 6
0152   C  BEFORE PROCESSING.
0153
0154         J = 1
0155         DO 350 I = 1,8
0156         CALL EXEC(1,DLU,FILDIR,128,DRTRK1,TEST)
0157         IF(FILDIR(J).GE.0) GO TG 270
0158         GO TO 340
0159    270  CONTINUE
0160         IF(FILDIR(J).EQ.0) GO TO 9990
0161
0162   C  DETERMINE IF THERE IS A BIT SET IN THE UPPER BYTE OF WORD FIVE.
0163   C  IF SO, THEN DETERMINE IF THE FILE SHOULD BE PROCESSED
0164
0165         L = J
```

```
 PAGE 0004  CLEXT  8:42 AM  TUE., 20  MAY , 1980


0166          L = L+5
0167          IEXT = IAND(177400B,FILDIR(L))
0168          IF(IEXT.EQ.0) GO TO 340
0169
0170    C  DETERMINE IF THE EXTENT SIZE IS THE DESIRED FILE SIZE SPECIFIED
0171    C  WHEN PROGRAM WAS RUN. IF SO, THEN SKIP THE FILE.
0172
0173          L = L+1
0174          IF (FILDIR(L)/2.EQ.SIZE) GO TO 340
0175
0176    C  READ NAME OF THE FILE THAT CONTAINS EXTENTS AND NOTIFY THE USER
0177
0178          M = J+2
0179          K = 3
0180
0181    C  PLACE THE FILE NAME INTO (CRNTFL)
0182
0183          DO 280 N = 1,3
0184     280  CRNTFL(N) = FILDIR(J+N-1)
0185
0186    C  OBTAIN THE SECURITY CODE OF THE FILE
0187
0188          SECODE = FILDIR (J+8)
0189
0190    C  DETERMINE IF CLEAR EXTENTS, OR SET TO A SPECIFIED SIZE.
0191
0192          IF(SIZE.EQ.-1) GO TO 300
0193          WRITE (LISTLU, 290)(CRNTFL(N),N = 1,3)
0194     290  FORMAT (" PROCESSING FILE ",3A2)
0195          GO TO 320
0196
0197     300  WRITE(LISTLU,310)(CRNTFL(N),N = 1,3)
0198     310  FORMAT (" CLEARING EXTENTS FOR " 3A2)
0199
0200    C  PROCESS THE FILE HERE
0201
0202     320  ERROR = XTNTS (CRNTFL,SECODE,CRNUM,SIZE,ERR)
0203
0204    C  CHECK FOR AN ERROR WHILE PROCESSING THE FILE
0205
0206          IF(ERROR.GE.0) GO TO 340
0207          WRITE (LLU,330)(ERROR,CRNTFL)
0208     330  FORMAT (" FMP ERROR "I3" WHILE PROCESSING FILE "3A2)
0209          GO TO 9990
0210
0211     340  J = J+16
0212     350  CONTINUE
0213
0214    C  DETERMINE IF THE PRESENT DIRECTORY TRACK HAS BEEN COMPLETELY
0215    C  SEARCHED.  IF SO, DETERMINE IF THERE IS ANOTHER DIRECTORY TRACK
0216    C  AND IF SO CONTINUE WITH THE SAME PROCESS.  IF NOT, BYE - BYE!
0217
0218          COUNTR = COUNTR+1
0219          IF(COUNTR.EQ.SECTRK/2) GO TO 360
0220          GO TO 240
```

```
 PAGE 0005  CLEXT  8:42 AM  TUE., 20  MAY , 1980


0221
0222    360  CONTINUE
0223         TKCNTR = TKCNTR+1
0224         IF(TKCNTR.GT.NMDTRK) GO TO 9990
0225         DRTRK1 = DRTRK1-1
0226         GO TO 230
0227
0228    9990 CONTINUE
0229
0230    9999 CONTINUE
0231
0232         END


  FTN4 COMPILER: HP92060-16092 REV. 2001 (791101)

  **  NO WARNINGS **  NO ERRORS **   PROGRAM = 00975      COMMON = 00000
```

```
 PAGE 0006  FTN.   8:42 AM  TUE., 20  MAY , 1980


0233          FUNCTION XTNTS (NAME,SC,CRN,SIZE,ERR)
0234          IMPLICIT INTEGER (A-Z)
0235    C
0236    C     This function will store the file called 'NAME' into a
0237    C     scratch file of 'SIZE' blocks per extent. When the
0238    C     transfer is complete, the file 'NAME' will be purged, and
0239    C     the scratch file will be renamed to 'NAME'.
0240    C     If there is an error at any point along the way, 'NAME' will
0241    C     be left unchanged, and the FMP error will be returned both as
0242    C     the value of the function, and as 'ERR'. If an error occurs
0243    C     after NAME is purged, a message will be written to LU 1 indicating
0244    C     the current state of affairs, and an error message will be printed
0245    C
0246    C     If the file is a type 1 or 2 file, XTNTS will be 0, and the files
0247    C     will be unchanged.
0248    C
0249    C     Programmer: Clark Johnson
0250    C     Date: 2-21-80
0251
0252          DIMENSION DCBA (144),                DCBB (144)
0253          DIMENSION NAME (3),                  SCRTCH (3)
0254          DIMENSION BLOCKS (2),                RECLEN (1)
0255          DIMENSION BUFFR (200),               BLEN (1)
0256          DIMENSION NEWERR (25),               NAMFER (21)
0257          DIMENSION NAMFEB (14)
0258
0259          EQUIVALENCE (NEWERR (23),SCRTCH)
0260
0261          DATA BLEN /200/
0262
0263          DATA NEWERR /2HAl,2Hl ,2Hsc,2Hra,2Htc,2Hh ,2Hfi,2Hle
0264         &,2H n,2Ham,2Hes,2H u,2Hse,2Hd.,2H L,2Has,2Ht ,2Hsc,2Hra,2Htc,2Hh
0265         &,2H= ,2H  ,2H  ,2H  /
0266          DATA NELEN /25/
0267
0268          DATA NAMFER /2HRe,2Hna,2Hme,2H e,2Hrr,2Hor,2H, ,2HNA,2HME,2Hxx
0269         &,2H i,2Hs ,2Hno,2Hw ,2Hca,2Hll,2Hed,2H: ,2HSC,2HRT,2HCH/
0270          DATA NALEN /21/
0271
0272          DATA NAMFEB /2HEx,2Hte,2Hnt,2Hs ,2Hha,2Hve,2H b,2Hee,2Hn ,2Hcl
0273         &,2Hea,2Hre,2Hd ,2Hup/
0274          DATA NALENB /14/
0275
0276          DATA SCRTCH /2H.X,2HTN,2H00/,        LASTN /2HEN/
0277
0278          LU = 1
0279          WRIT = 2
0280          BLOCKS (1) = SIZE
0281          OPTN = 0
0282
0283    C     Open the file 'NAME'
0284              CALL OPEN (DCBA,ERR,NAME,OPTN,SC,CRN)
0285                  XTNTS = ERR
0286                  IF (ERR .LE. 0) RETURN
0287
```

```
 PAGE 0007  XTNTS  8:42 AM  TUE., 20  MAY , 1980

0288                TYPE = ERR
0289
0290  C             If the file is type 1 or 2, get out
0291                    IF (TYPE .NE. 1) GO TO 100
0292                    IF (TYPE .NE. 2) GO TO 100
0293
0294  C                 ELSE
0295                        CALL CLOSE (DCBA)
0296                        XTNTS = 0
0297                        RETURN
0298
0299   100  CONTINUE
0300
0301            BLOCKS (2) = RECLEN
0302            CALL CREAT (DCBB,ERR,SCRTCH,BLOCKS,TYPE,SC,CRN)
0303                IF (ERR .GT. 0) GO TO 200
0304
0305  C             ELSE  (handle some errors, return on the rest)
0306                    XTNTS = ERR
0307                    IF (ERR .NE. -2) RETURN
0308
0309  C                 ELSE (duplicate scratch file name, so get new name)
0310                        IF (NEWNM (SCRTCH) .NE. LASTN) GO TO 100
0311
0312  C                     ELSE (NEWNM has run out of names to try so..)
0313                            CALL EXEC (WRIT,LU,NEWERR,NELEN)
0314                            XTNTS = ERR
0315                            RETURN
0316
0317   200          CONTINUE
0318
0319  C     Transfer records from A to B
0320            CALL READF (DCBA,ERR,BUFFR,BLEN,LEN)
0321                IF (ERR .LT. 0) GO TO 1111
0322
0323                IF (LEN .EQ. -1) GO TO 900
0324
0325            CALL WRITF (DCBB,ERR,BUFFR,LEN)
0326                IF (ERR .LT. 0) GO TO 1111
0327
0328  C     Loop for next record
0329            GO TO 200
0330
0331  C     EOF found, so transfer is complete.
0332   900      CONTINUE
0333
0334  C     Purge 'NAME'
0335            CALL PURGE (DCBA,ERR,NAME,SC,CRN)
0336                IF (ERR .LT. 0) GO TO 1111
0337
0338  C     Close 'SCRTCH'
0339            CALL LOCF (DCBB,ERR,I,NXTBLK,I,SZE)
0340
0341                IF (ERR .LT. 0) GO TO 1111
0342
```


```
 PAGE 0008  XTNTS  8:42 AM  TUE., 20  MAY , 1980


0343  C              calculate number of blocks to delete from file
0344                     TRUNC = SZE/2 - NXTBLK - 1
0345                     IF (TRUNC .LT. 0) TRUNC = 0
0346
0347             CALL CLOSE (DCBB,ERR,TRUNC)
0348
0349  C      Rename 'SCRTCH'
0350             CALL NAMF (DCBB,ERR,SCRTCH,NAME,SC,CRN)
0351                 IF (ERR .GE. 0)  GO TO 950
0352
0353  C              ELSE (error on rename of scratch file)
0354                     DO 930 J = 1,3
0355                         NAMFER (7+J) = NAME (J)
0356                         NAMFER (18+J) = SCRTCH (J)
0357   930               CONTINUE
0358
0359                     CALL EXEC (WRIT,LU,NAMFER,NALEN)
0360                     CALL EXEC (WRIT,LU,NAMFEB,NALENB)
0361                     XTNTS = ERR
0362                     RETURN
0363
0364   950           CONTINUE
0365
0366  C      Everything should be closed now.
0367             XTNTS = ERR
0368             RETURN
0369
0370  C      Error handling section for READF, WRITF, and PURGE calls
0371    1111     CONTINUE
0372
0373                 CALL CLOSE (DCBA)
0374                 CALL CLOSE (DCBB)
0375                 CALL PURGE (DCBB,ERROR,SCRTCH,SC,CRN)
0376                 XTNTS = ERR
0377                 RETURN
0378         END


  FTN4 COMPILER: HP92060-16092 REV. 2001 (791101)

  **  NO WARNINGS **  NO ERRORS **   PROGRAM = 00846      COMMON = 00000
```

```
 PAGE 0009  FTN.   8:42 AM  TUE., 20  MAY , 1980


0379         FUNCTION NEWNM (NAME),Function to generate 6 char names
0380         IMPLICIT INTEGER (A-Z)
0381   C
0382   C     NEWNM will take a 6 character file name, and return a
0383   C     NEW 6 character file name. It is useful for generating
0384   C     scratch files.
0385   C
0386   C     It will take a name in the form,   xxxx00
0387   C
0388   C     where          xxxx       are any 4 filename characters
0389   C     and            00         are two digits between 0 and 9.
0390   C
0391   C     The file name will be modified by adding 1 to the number
0392   C     at the end of the name.
0393   C     i.e.   TEST00  will become  TEST01 and so on until TEST99
0394   C     is reached.
0395   C
0396   C     When xxxx99 is reached, the next time the function is called,
0397   C     NEWNM will be set to EN (for END), and the name will be unchanged.
0398   C
0399   C
0400   C     Programmer : Clark Johnson
0401   C     Date: 2-21-80
0402
0403         DIMENSION NAME (3)
0404
0405         DATA LASTN /2HEN/,          BLANKS /20040B/
0406
0407   C **** FUNCTIONS **************
0408
0409         RBYTE (J) = IAND (J,377B)
0410
0411   C **** BEGIN THE CODE *********
0412
0413         NEWNM = BLANKS
0414
0415   C     Is NAME in range?  (xxxx00 or greater)
0416               IF (NAME (3) .GE. 30060B) GO TO 200
0417
0418   C           ELSE (out of range)
0419                    NEWNM = LASTN
0420                    RETURN
0421
0422    200            CONTINUE
0423
0424   C     Has the last name been generated?  (xxxx99 or less)
0425               IF (NAME (3) .GE. 34471B) GO TO 300
0426
0427   C           ELSE ( not at end of names,so generate a new one)
0428                    IF (RBYTE (NAME(3)) .GE. 71B) GO TO 250
0429
0430   C                ELSE (last digit is less than 9)
0431                         NAME (3) = NAME (3) + 1B
0432                         RETURN
0433    250             CONTINUE
```

```
 PAGE 0010  NEWNM  8:42 AM  TUE., 20  MAY , 1980

0434
0435  C              ELSE (Must roll over and increment 10s column)
0436                     NAME (3) = NAME (3) + 367B
0437                     RETURN
0438
0439   300       CONTINUE
0440                     NEWNM = LASTN
0441                     RETURN
0442             END


  FTN4 COMPILER: HP92060-16092 REV. 2001 (791101)

  **  NO WARNINGS **  NO ERRORS **   PROGRAM = 00092      COMMON = 00000
```

HP SUBROUTINE LINKAGE CONVENTIONS

Robert Niland/HP Lexington

[Editor's note: This is the fifth part in a series of articles taken from Bob Niland's Links/1000 contributed manual. The articles explain subroutine linkage conventions for RTE-IV, and much of the information can be applied to RTE-II and RTE-III as well. For a definition of the MPX macro instruction see the first article in the series (Volume III, Issue 6).]

6-1. WHAT IS A PRIVILEGED ROUTINE?

Ordinarily, your application code executes with the computer's interrupt system enabled (on). Interrupts may occur during your program, and come from a variety of sources, such as:

a. I/O devices, which may or may not be performing an operation on behalf of your program.

b. Internal computer trap cells, such as parity error, or power fail.

c. Program events, such as an EXEC call, MP/DM violation etc. This may be intentional or unintentional.

Let us consider Memory Protect (MP) violations. The Memory Protect capability of the computer is the operating system's primary means of protecting itself from modification (corruption) by a user's program. Any attempt to modify a location below the address currently loaded in the fence register on the 12892B card will result in an interrupt on select code05.

Since the JSB instruction writes the return address into the destination entry point, it will generate an interrupt if that entry point (EXEC for example) is below the fence. This is true even if you were attempting a legal operation (such as an EXEC call). The interrupt causes the execution of the instruction stored by RTxGN in trap cell 05, usually a JSB through a base-page link to $CIC or $CIC0. CIC (Central Interrupt Control) looks in its own entry point to see who was interrupted (you), and in the CPU interrupt register (LIA 04) to see what caused the interrupt (in this case your JSB). CIC examines your destination and calling parameters, and if legal, passes control to the appropriate modules in the RTE resident code (which, by the way, may or may not be the address of your JSB). If your activity is not legal, RTE will abend your program with an MP error.

There are occasions when it is necessary for an HP subroutine appended to your program, or even your program itself, to perform tasks for which there is no standard system call, and which would normally result in an MP error. These tasks must therefore be performed with the interrupt system off.

Code which executes with the interrupt system off is called privileged. It should be pointed out, however, that privileged user code is not the same as a privileged driver, although the concepts are similar. And, since privileged drivers operate entirely outside RTE, they form no part of the subroutine linkage conventions, and will not be treated in this manual.

Extreme caution must be used within privileged code. The following should be kept in mind:

1. All interrupts except Power-Fail, Parity Error, and any Privileged I/O are held off. If your code takes longer than 1 millisecond to execute, characters may be lost on some types of terminal interfaces, and if held off longer than 10 ms., one or more time-base interrupts (clock ticks) will certainly be lost.

2. Coding errors which would MP abend a normal program may crash or subtlely corrupt the RTE operating system if executed while privileged.

A privileged subroutine is one which turns off the interrupt system upon entry. The generator recognizes these modules by their type 06 NAM records, i.e. NAM SUBPR,06 optional comment They can only be written in assembly language, and require no special knowledge or action on the part of the calling program.

There is another class of subroutines, usually found in the RTE system library (%4SYLB,%MSYLB, etc), which have as part of their calling sequence in the RTE Programmer's Reference Manual the requirement that the caller must be privileged while calling. This is not the same as a privileged subroutine. What it means is that the caller must have already turned the interrupt system off before calling the routine. This type of routine is usually resident in the operating system itself. Simply put, it is not a privileged routine. It is called by privileged routines.

6-2. WHAT IS A RE-ENTRANT ROUTINE?

When an interrupt occurs, the state of your program is saved by the operating system. This consists of recording in your program's ID Segment the point of suspension (P), and the working registers (A,B,E,O,X,Y). It does not include saving the current state of any variables, buffers, or modified code within your program, except that the entire program may be "saved" by the process of swapping it out to disc if another program has priority access to your partition.

The fact that the state of your code is not saved is not normally of any concern. Although you may have issued the same

```
:RU,EDITR
```

command as one or more other users on the system, you are each executing your own copy of the code. For example, most of this manual was written while the author was running EDI81. And although many programmers may call RMPAR in their applications programs, each program has its own copy of RMPAR appended to it.

There are circumstances in which two users might be contending for the use of the same code (typically a subroutine). For example, suppose we have two memory resident programs which must compute standard deviations, and for considerations of space in the memory resident area, we want to have only one resident copy of our subroutine (called SIGMA). Both programs will be linked to SIGMA by RTxGN. At some point program A (PROGA) is executing and calls SIGMA. Suppose that within SIGMA we are at the point where we are computing N-1:

```
...
FAD = D-1       *Compute N-1.
DST N.TMP       *Save in temporary location.
```

and an interrupt occurs. PROGB has been scheduled (by event, clock, operator, or whatever), and it has a higher priority. PROGA gets suspended, its registers get saved, but the variables in SIGMA, such as N.TMP do not get saved. PROGB's call to SIGMA would run to completion, return, and PROGB would either terminate or suspend on some system resource (such as I/O). PROGA would be restarted at the point of suspension. This means that PROGA's registers would be restored, but the value of variables within the shared code of SIGMA would be now be set to whatever they were when SIGMA returned to PROGB!

To prevent the kind of conflict described on the previous page we need to write SIGMA in such a way that:

1. PROGB cannot interrupt SIGMA (and PROGA) until SIGMA has run to completion.

   –or–

2. SIGMA saves (or asks RTE to save) the state of its variables on entry, and restores them on exit, so that it map be interrupted with no ill effects.

Solution 1 is known as a privileged subroutine, just as in section 6-1. This is simply another reason for it.

Solution 2 is known as a re-entrant subroutine, because the code can be re-entered while another program is still in it.

To summarize sections 6-1 and 6-2:

Privileged subroutines:

1. Execute with the interrupt system off.

2a. Are used when the caller needs to do something which is normally illegal.

—and/or—

2b. The subroutine needs to run to completion without being interrupted.

Re-entrant subroutines:

1. Execute with the interrupt system on.

2. Are used when the subroutine:
   a. Has more than one potential simultaneous caller.
   b. Takes longer to execute than is wise using privileged mode.

6-3: PRIVILEGED MODE WITH $LIBR AND $LIBX

Privileged routines are distinguished from ordinary routines by the fact that they execute with the interrupt system off in normal systems, and above the 12620A Privileged Interrupt Card in systems with privileged I/O subsystems. This section deals with the question:

How do I turn the interrupt system off?

The way it is not done is to write an ASMB subroutine which contains a CLF 00 instruction. Attempting to execute an I/O instruction while the interrupt system is on will result in your program being abended by RTE with an MP error. (Operations on select code 01, the O & S registers, are excepted.)

Instead you must request that RTE turn it off for you. This is done by calling a special system entry point. This entry point is below the fence, so an MP interrupt (and its overhead) will be generated, but if you call it correctly, RTE will return to you with the interrupt system off. It is also used for re-entrant subroutines, in which case the interrupt system will be on, and that is discussed in section 6-4.

This entry point is $LIBR. Naturally, when your routine has completed, you will want to turn the interrupt system back on, so $LIBR has a complement, $LIBX, which does this.

$LIBR and $LIBX can be used for two kinds of privileged routines:

1. For subroutine entry/return, a task for which they are optimzed.

2. For "in-line" code where you just want to go privileged for a few instructions (or calls), and where you do not want to write that code as a separate subroutine.

We will discuss use 1 first, since it is the most straight-forward. We will also discuss use 2, showing how to "bend" $LIBX's desire to return through your code rather than just to it.

We will explain $LIBR/$LIBX protocol first for an elementary type subroutine, i.e. one to which parameters are being passed only in the registers. Be advised that $LIBR and $LIBX do save and restore all the working registers for you.

The sequence is:

```
        NAM SUBPR,06  Comment for RTxGN/LOADR load map.
        ...
        ENT SUBPR          Need not be NAM, must = entry point name.
        ...
        EXT $LIBR,$LIBX    Are in TB-I or EXEC.
        ...
SUBPR   MPX                Entry point/return address.
        JSB $LIBR          Turn off interrupt system,
        NOP                and remain privileged.
        ...
        ...                Execute while privileged.
        ...
        JSB $LIBX          Turn interrupts back on,
        DEF SUBPR          and return to caller through entry point.
```

The alert reader may ask: If this is a resident library routine, and is privileged to avoid re-entrant conflicts, what happens if PROGA's JSB has written the return address into SUBPR, but the JSB $LIBR hasn't executed yet, and the higher priority PROGB gets scheduled and calls SUBPR? Won't PROGA's return address get lost?

The answer is no. Depending on which RTE we are discussing, one of three things will inhibit PROGB's request until beyond the JSB $LIBR, that is, until PROGA's invocation safely completes.

1. Everyone's JSB SUBPR may be indirect through a current or base- page link. The interrupt system is inhibited on any JMP or JSB-indirect until the next instruction (at entry point +1) is executed.

2. The Memory Resident Library may be below the MP fence, in which case the JSB $LIBR is superfluous, as RTE will enter the privileged routine, with interrupts off.

3. The Memory Resident Library may reside on a write-protected page, in which case entry is also from RTE and not through the actual SUBPR entry point.

To summarize the rules for a simple privileged routine:

1. Tell RTxGN that this is a privileged routine with a 6 in the NAM record. Since shareable code can only be generated into the system, LOADR will treat type 6 modules as type 7.

2. Place a JSB $LIBR in the next memory location after the subroutine's entry point. Any intervening code will not cause $LIBR or $LIBX to fail, but if the subroutine is shared, it may be interrupted between the entry point and the JSB $LIBR resulting in loss of the first caller's return address.

3. Place an all-zeros word in the memory location after the JSB $LIBR. This is what distinguishes a privileged $LIBR call from a re-entrant one.

4. Execute your privileged code, and:
   a. Do not call any but another privileged routine. In particular do not call any re-entrant routines.

   b. Do not make any EXEC request, including I/O, or you will either be abended with an EX error, or in earlier RTE's, you will halt RTE.

   c. Do not attempt recursion. You may not call yourself, for among other reasons:
      I. You will destroy your return address.
      II. Privileged routines can only be nested to a level of 32767, and you would consume too much time far before reaching that level.

d. Do not fool around with I/O trap cells, interface cards, or the interrupt system. Yes, if you issue a HLT 00, you will actually halt the computer.

5. You can, at your own risk, read from and write to locations below the MP fence. You can perform cross-map reads from and writes to locations in the alternate (system) map. You may not read or write locations in physical memory which are not in your map or the system map. To do this, you will need to alter the contents of your mapping registers. However, unless you are very familiar with mapping under RTE, you'd best put the registers back the way they were prior to exit.

6. Place a JSB $LIBX after the last executable instruction in your subroutine. When executed, RTE will return control to your caller with the interrupt system back on, unless the caller was also privileged, in which case it will be left off.

7. Place a DEF <entry point name > in the next memory location following the JSB $LIBX. This DEF may be indirect as long as it eventually points to the entry point.

A privileged routine need not be a privileged subroutine. It is possible for an ASMB program or subroutine to contain a few lines of code which need to execute while privileged. The principal obstacle to writing "in-line" privileged code is that $LIBX expects to perform a subroutine return, and returns to your code in a "double-indirect" fashion. There is an address following the JSB $LIBX which points to a location containing the return address.

Your in-line code can handle this in the following fashion:

```
        ...
        ...             Normal non-privileged code.
        ...
        JSB $LIBR       Turn off interrupts,
        NOP             and remain privileged.
        ...
        ...             Execute while privileged.
        ...
        JSB $LIBX       Restore the interrupt system,
        DEF @RESU       and return through RET,I
        ...
        ...             Non-executable memory locations (optional).
        ...
@RESU   DEF RESUM       Address of resumtion of normal execution.
        ...
        ...             More non-executing locations (optional).
        ...
RESUM   ...             Non-privileged execution resumes.
```

The locations DEF RESU, RESU DEF RESU, are usually adjacent, and could also be expressed as:

```
        ...
        ...
        JSB $LIBX       Restore the interrupt system,
        DEF *+1         and return through next word,I
        DEF *+1         return to next location.
        ...             Resume non-privileged execution.
```

This mode of operation is possible because, unlike .ENTR, $LIBR does not require an entry point nearby. And, $LIBX only requires a resumption point link address.

For an example of simple privileged code, refer to the listing of routine PRIOR in Appendix F. This is not a privileged subroutine, but does contain some privileged code.

6-4. RE-ENTRANT MODE WITH $LIBX AND $LIBR

Re-entrant routines are distinguished by the fact that one resident copy of their code, while servicing a caller, can be interrupted and called by another program. This requires that temporary data relating to the first caller be saved and restored.

The data saving and restoring is done for you by RTE, using a variation on the privileged $LIBR/$LIBX call. The difference is that you must tell RTE that you want to be re-entrant rather than privileged, and you must tell it where the data is, and how much you have.

The sequence is:

```
        NAM SUBRE,06  Comment for RTxGN/LOADR load map.
        ...
        ENT SUBRE             Need not=NAM, must=entry point name.
        ...
        EXT $LIBR,$LIBX
        ...
SUBRE   MPX                   Entry point/return address.
        JSB $LIBR             Turn off interrupt system briefly,
        DEF TDB               and save previous T.D.B. (if any).
        ...
        ...                   Execute while re-entrant.
        ...
        JSB $LIBX             Turn off interrupts briefly,
        DEF TDB               restore the previous TDB,
        DEC <ret>             and return to SUBRE,I or SUBRE,I+1
        ...
TDB     NOP                   TDB link list header.
        DEC <tdb>+3           Size of TDB data area + header.
        NOP                   Return point used by $LIBX.
DATA    BSS <tdb>             Actual data storage area of <tdb> words.
        ...
```

The rules are:

1. The NAM record must be type 6. As for privileged routines, it tells RTxGN that if one or more Memory Resident programs require this routine, only one copy of it needs to be relocated into the memory resident library.

2. Place the JSB $LIBR in the memory location following the entry point.

3. Place the address of the Temporary Data Block (header plus data areas) in the location following the JSB $LIBR.

4. While within the re-entrant routine:

   a. Do not modify any memory locations outside those in the TDB.

   b. That includes not modifying any instruction within your routine. This means that you may not use HP1000 instructions which are interruptable and which save the current loop index in a location adjacent to the instruction (initially a NOP) when interrupted (for example, CBT, CMW, and MVW).

   c. Do not call any "library" (type 7) subroutines. These are neither privileged nor re-entrant, and if you are interrupted while within such routines, your copy of their temporary variables will be lost.

5. While within the re-entrant routine, you may:

   a. take as long as necessary.

   b. perform I/O (but not with "library" device subroutines).

   c. call other privileged or re-entrant routines.

   d. modify data in the TDB. You access it like any local storage/variables in any ASMB routine. In fact the BSS form of declaration is really not the best. A more elegant form is the symbolic:

```
        ...
TDB     NOP             The same.
        ABS EOD-TDB     Always correct, even if data altered.
        NOP             The same.
DATA1   MPX             The first variable name/location.
ARRAY   BSS 24          An array perhaps?
DEX     BSS 3           An extended precision result.
EOD     EQU *           The next location (outside TDB).
        ...
```

6. Place a JSB $LIBX after the last executable instruction in the routine.

7. Place the address of the local Temporary Data Block (DEF TDB) in the location following the JSB $LIBX.

8. Follow step 7's address with a location containing a 0 or 1:

   0: If the subroutine (SUBRE) makes a simple return always.

   1: If the subroutine will make a simple return on an error, but will skip one location in the caller if returning without error. An error might occur if, for example, RTE would never have enough System Available Memory in which to store the TDB.

9. Build the TDB header somewhere in the routine, consisting of:

   a: An all-zeros location in which RTE puts the linked-list pointer to previously saved TDB's. Do not modify this location in your routine. RTE will set it back to zero when appropriate. It must be followed by...

   b. A location containing the size of the TDB, which is 3 plus the number of words in the data area. This must be followed by...

   c. Another all-zero location in which RTE will save the return address from your entry point. (Otherwise the next caller would destroy it.)

10. The TDB header must be immediately followed by the TDB data area, consisting of any number of NOP's or pseudo-instructions necessary to allocate the quantity of storage space needed by the routine. Each element may have its own label. Constants should not be placed in this area. They will be needlessly "swapped-out" to SAM with the variables when re-entry occurs.

When to Make a Subroutine Privileged or Re-Entrant:

If you are attempting something which is normally illegal, the need for a privileged routine is apparent. It is not so obvious in other cases. The $LIBR $LIBX entry exit protocol should only be used when the subroutine is likely to be called by one or more memory-resident programs, and the space/performance cost of making it privileged/re-entrant is lower than the space cost of having one copy of the routine for each program.

FUNCTIONS OF A BAIMG

Carol Jonas/HP Data Systems Division

The BASIC/IMAGE Interface is more than just extra overhead for doing data base manipulations from BASIC programs. It performs many functions necessitated by differences between BASIC/1000D and other HP/1000 languages as well as resolving the fact that the BASIC Interpreter can only interpret BASIC code. It is the purpose of this paper to explore those differences and to exemplify the services provided by the BASIC/IMAGE Interface.

The BASIC/IMAGE Interface works on the overlay principle in BASIC/1000D. BASIC overlays are used to allow a BASIC program to call subroutines written in another language. Each Basic overlay is just a separate program which is scheduled by BASIC. Parameters are passed back and forth between BASIC and the overlay through class I/O (SAM). A more detailed explanation of the passing of parameters between BASIC and an overlay can be found in the HP/1000 HPIB Application Note 201-8 HP part number 5953-4216.

In order for a BASIC program to use other language subroutines, the overlay for the subroutines must be prepared by another program, the BASIC table generator. This program performs many functions related to building the overlays and preparing information for their use by BASIC. The main program for each overlay is actually created by the BASIC table generator, RTETG. The main merely consists of a program name and the external references necessary to bring in, at load time, the subroutines being interfaced to BASIC.

The only other principle part of each overlay supplied by BASIC is CALSB (short for call subroutine). It is CALSB's function to retrieve and store each parameter passed by BASIC into the overlay's address space, to perform the subroutine call and to pass back each return parameter from the overlay to BASIC. CALSB finds space to store the parameters passed by BASIC by using the words remaining in the partition from the last word of the overlay code to the last word of the partition. For this reason, each BASIC overlay must be allocated extra page(s) using the SZ command in RTETG, the loader or the system.

When RTETG is run, it must be given explicit details on the type of each parameter used by the subroutine(s) in the overlay. BASIC has only two types of data formats: real and character string. BASIC's real data format is the same as the machine representation of a two-word real and is therefore the same as a real in any other language. However, BASIC's character string format does not conform to any other language's. Its character strings are one dimensional arrays with ASCII characters starting in the third byte of the array preceeded by two bytes of length information. The first byte contains the length in bytes of the physical storage area occupied by the character string. The second byte contains the number of characters (bytes) that have been entered into the string. For example, assume a character string variable declared as:

```
DIM A$(10)
```

and initialized by:

```
LET A$ = "STRING"
```

then the first byte would contain the value 10 and the second byte would contain the value 6.

In order for a subroutine of another language to use data passed to it by BASIC, that data must be in a format that it recognizes. For this reason, parameters passed by a BASIC program to an overlay subroutine may need some form of conversion. The only data format that does not need to be converted is the real format. It can be passed back and forth and used by either BASIC or the overlay subroutine as is. BASIC itself provides a limited conversion capability. It can convert between real and integer format for the overlay, if told to do so. By specifying the type of data for each parameter in the subroutine to RTETG, it can build what are called the Branch and Mnemonics Tables to inform BASIC of the data formats required by the subroutine(s) in the overlay. Before scheduling the overlay, if necessary and possible, BASIC transforms the parameters from its format into the format required by the subroutine(s) and on return, back into its own format. For example, if a subroutine requires a machine format single word integer parameter, BASIC will convert a real parameter into an integer before writing it to the class, and if the subroutine returns an integer parameter, BASIC will convert that integer into a real.

Any subroutine in which each parameter has only one possible simple numeric type (integer or real) may be called directly from CALSB in the overlay since BASIC performs these conversions. However, many subroutines, particularly the IMAGE subroutines, may have parameters of other data types and may also allow the type of some of the parameters to vary which requires an interface between CALSB and the true subroutine. For instance, the argument parameter in a DBGET call can be used to hold a double integer record number or a real, integer or FORTRAN format character string data item value.

Another concern is record type data formats. BASIC does not allow mixed mode arrays. Each entry in an IMAGE data set may contain many data fields of varying type. A FORTRAN program may receive an entire entry as a mixed mode array and proceed to process each field in the array based on its type. However, a BASIC program can only work with different variables and arrays each of a specific type and they may not be subfields of a larger mixed mode array.

For the IMAGE subroutines, and any other subroutine with similar features, extra data conversion must be performed before they can use the parameters passed by BASIC. The BASIC/IMAGE interface, BAIMG, performs this extra data conversion for the IMAGE subroutines.

BAIMG is a set of subroutines itself. This set of subroutines is appended to the overlay main created by RTETG at load time. It is BAIMG which contains the entry points DMOPN, DMCLS, etc. which are specified in the required answer file for RTETG when describing the BASIC/IMAGE overlay. It is these entry points which are entered by CALSB when it performs the subroutine calls to the IMAGE routines. Each of these DMXXX routines then performs the extra data conversion, calls the corresponding DBXXX routine, and upon return, performs any reverse data conversion necessary.

The kind of extra data conversion performed by BAIMG can best be exemplified by the DBGET (DMGET) routine. The RTETG command describing the DBGET calling sequence is a follows:

```
DBGET(RA,RA,I,RVA,RA,RA,RVA,RVA,RVA,RVA,RVA,RVA,RVA,RVA),
      VL,OV=nn,SZ=mm,ENT=DMGET,FIL=%BAIMX
```

For a detailed explanation of the components of RTETG commands please see the BASIC/1000D Programmer's Reference Manual HP part number 92060-90016. For the purposes of this paper it is sufficient to define the parameter descriptors. Each descriptor is composed of the letters R, I, V and A in some arrangement. R means that the parameter is a real. Real parameters need no conversion by BASIC. I means that the parameter is an integer. Parameters of type integer in an overlay subroutine must be of numeric type in the BASIC program. BASIC will convert the real to an integer for the overlay. V means that the parameter is to receive a value upon return to BASIC. If a parameter descriptor contains an I as well as a V, BASIC will convert the returned value into a real. A means that the parameter is an array. BASIC can perform real array to integer array conversion and vice versa. Note that there is no way of signifying character string parameters. All character string parameters can be passed to the overlay as real arrays (RA) or returned as real value arrays (RVA) but they are passed in BASIC format, length fields and all.

DBGET has seven parameters, in the following order, all of them required:

ibase	is a data base descriptor
id	is a data set descriptor
imode	describes the method of reading desired by the caller
istat	is a status array
list	is a data item descriptor list
ibuf	is a buffer for returned data item values
iarg	is the argument parameter discussed above

DMGET has fifteen parameters, in the following order, only seven of which are required:

ibase	
through	are as above
istat	
iarg	is as above
name-list	is as list above
value-list	is a set of one to nine variables in which the data item values are returned

When a BASIC program calls DBGET it uses the name DBGET but the calling sequence is that of DMGET. In order for BAIMG to perform the actual DBGET call, the parameters in the DMGET calling sequence must be transformed into the DBGET calling sequence. Aside from order, many of the parameters have limited or differing meanings. Taking the parameters one at a time, DMGET does the following.

ibase — Corresponds to the first parameter descriptor, RA, in the RTETG command and as such is not touched by BASIC before being passed to the overlay. This parameter must be a character string in the BASIC program. DMGET increments the address of the parameter by one to skip over the length bytes in the first word then passes this new address as the address of the ibase parameter to DBGET.

id — Corresponds to the second parameter descriptor, RA, in the RTETG command and as such is not touched by BASIC before being passed to the overlay. For BASIC this may only be a character string containing a data set name (from FORTRAN a data set number is allowed to DBGET). However, it must be a six character name. DMGET takes the actual characters specified in the string, as defined by the length count in the second byte of the array, and moves them into a new three word array padding them out to six characters with blanks as necessary. It is the address of this new array that is put in the DBGET calling sequence.

imode — Corresponds to the third parameter descriptor, I, in the RTETG command and, therefore, BASIC does the necessary real to integer conversion before passing the parameter to the overlay. DMGET passes the parameter on through to DBGET exactly as received from CALSB. No data movement or address correction is necessary.

istat — Corresponds to the fourth parameter descriptor, RVA, in the RTETG command and is not touched by BASIC before being passed to the overlay nor after it is returned. This parameter must be a real array of at least 6 elements within the BASIC program. To DBGET this parameter must be an array of at least 10 words in which mixed mode integer and double integer numbers will be returned. On entry to DBGET, no conversion is required because this is a return value only parameter. In fact, the status array passed to DBGET by DMGET is a ten word array internal to BAIMG rather than the parameter passed by CALSB. On return from DBGET, however, DMGET is responsible for converting any integers and double integers into reals and placing the real values into the parameter passed by CALSB in their proper order.

iarg — Corresponds to the fifth parameter descriptor, RA, in the RTETG command and as such is not touched by BASIC before being passed to the overlay. As sited earlier in this paper, this parameter may take on any data format. DMGET must determine, based on the imode and id parameters, and the iarg parameter itself, just what data format it is supposed to be. For instance, on a directed read, imode =4, DBGET will expect a double integer and the BASIC program will have to have passed a real. DMGET must therefore convert the real into a double integer. On a keyed read, imode =7, DBGET will expect a format that conforms to the format of the key item for the data set specified by id. DMGET determines the type of the key item by calling DBINF and then performs any necessary conversion. When iarg is to contain an integer data item value or a double integer record number, DMGET reuses the space occupied by iarg for the converted number. When a real or a character string is expected, iarg is passed as is, with an address correction, of course, if a character string.

name-list — Corresponds to the sixth parameter descriptor, RA, and as such is untouched by BASIC before being passed to the overlay. This parameter is always a data item name list as described by the list parameter for any IMAGE call. Therefore, it must be a character string in the BASIC program and is passed as is, with address correction for the length word, to DBGET in the list parameter location.

value-list — Corresponds to the seventh through fifteenth parameter descriptors, all RVA, and are untouched by BASIC before being passed to the overlay as well as after they are returned. This set of parameters are the tricky ones. Not only must data conversion be performed, but a record may need to be unpacked. DBGET is expecting to receive an array into which it may place possibly mixed mode fields at random. The array passed to DBGET as the ibuf parameter is internal to BAIMG. Upon return, this array must be broken into its separate fields, converted into reals or BASIC character strings, if necessary, and placed into the variables in the value list. Since all these variables are passed back as RVA to BASIC, BASIC will not do any conversion and will expect the variables to be in their proper format for the BASIC program's use. Note that for character string data items, this means that DMGET is responsible for setting up the length bytes in the return parameter also. Each variable in the value-list is supposed to be in one-to-one correspondence with the names in the name-list and are also assumed to be in the same order. DMGET determines the length of each field and its type from DBINF calls based on the item names in the name-list.

In short, DMGET, and all of BAIMG, is an intermediary between BASIC and IMAGE that must know both sets of calling sequences, both sets of data formats, and be able to convert freely both ways. The DMGET-DBGET interchange was chosen as an example of this capability since it is one of the most extensive, but each IMAGE subroutine requires extra data conversion to some degree.

As you can see, the data format incompatibilities is the major reason for the existence of the BASIC/IMAGE Interface. These incompatibilities not only result in extra data conversion but also in an alteration of the information returned by IMAGE calls in a BASIC program from the standard. For instance, all but one of the DBINF modes, when called from BASIC, return information in character strings that would normally be integer mixed mode integer, double integer and character. The two modes which return standard information leave the buffer containing it in a state that is uninterpretable by the BASIC program.

There are two more minor problems handled by BAIMG. One of these is the fifteen parameter limit imposed by BASIC on subroutines. The BAIMG routines DMGET, DMUPD and DMPUT must take this into account when packing or unpacking a data entry. No more than nine or ten fields of the entry, depending on the call, can be manipulated at a time. Also, the maximum length of a BASIC character string is 255 characters. Whenever passing character string data back to BASIC, the DMXXX routines must enforce this limit.

While the BASIC/IMAGE Interface was developed to resolve the differences between BASIC and other languages when using IMAGE, it does provide some benefits to the programmer. For one, having the IMAGE subroutine code in a different program, rather than appended to the BASIC program, allows for more code space for the BASIC program. Another major benefit is the lack of record types itself. Being able to access several data items at once as different variables relieves the burden on the programmer of breaking apart a possibly mixed mode array in order to manipulate the individual fields. These features, combined with the ease of programming in BASIC, make the BASIC/IMAGE Interface a valuable tool for data base applications.

FAST FORTRAN

John Pezzano/HP El Paso

Because it is a high-level language, FORTRAN has many advantages over Assembly. It is easier to write and debug, and is much more widely known. There are two major disadvantages of FORTRAN – speed and size. FORTRAN programs are invariably slower than their Assembly counterparts. The compiler generates less efficient code than the Assembly level programmer can write, and calls more library routines. The programmer has always had two choices – inefficient, easy FORTRAN, or efficient (and less well-known) Assembly. Now there is a third choice — efficient FORTRAN coding – what I call FAST FORTRAN.

FAST FORTRAN is a coding rather than a formalized programming technique, which includes:

1. Writing equations for efficient compilation
2. Understanding compiler inefficiencies and working around them
3. Understanding the limitations and advantages of system EXEC calls and library routines.

FAST FORTRAN is not a replacement for good programming techniques, but is an alternative to Assembly, where speed of execution and size of a program are important considerations. Here are some principles of FAST FORTRAN.

1. KNOW WHAT IT TAKES TO COMPILE AN EQUATION FOR MINIMUM COMPILATION

To understand this, one must understand the compiler. Each arithmetic or other operation creates a call to a library routine to accomplish the operation. The equation

```
X = Z**2 + Z
```

creates a call to .RTOR, the library routine to do real exponential and .FAD to add the two real numbers. On the other hand, this can be written as

```
X = Z*(Z + 1.0)
```

which creates only a call to .FAD and .FMP (real multiply). This is not only faster, but is implemented in microcode (in E- and M-Series computers) or in hardware (in F-Series).

2. REMOVE INVARIANT FROM A DO LOOP

Equations within a loop are executed multiple times. If there is any code within a loop that does not change, remove it when possible. In

```
      DO 10 = 1,000
      X(I) = SIN (X**2 + Y**2) + Z(I) * 2
   10 CONTINUE
```

SIN (X**2) does not change, yet recalculated N times. Replacing the above with

```
      R = SIN (X**2 + Y**2)
      DO 10 I = 1,000
      Z(I) = R + Z(I) * 2
   10 CONTINUE
```

saves 5 words, but the loop is only 18 words instead of 38, saving the execution of 2000 words for a 100X loop including 99 more "SINE" calls and 200 "RTOR" calls.

3. MAKE SURE CONSTANTS AGREE IN TYPE WITH VARIABLES

The equation

```
X = Y + 1
```

causes the compiler to make a call to "float" to convert 1 to a real.
The use of

```
X = Y + 1.0
```

saves that call. If Y is double precision, the cost is 4 words for "1" and 6 words for 1.00. While this may not seem like much, 50 mistyped constants in a program add 200 – 300 instruction words to a program.

4. COMBINE CONSTANTS

The equation – CIRCUMFERENCE = 2 PI RADIUS – which can be written

```
DATA PI/3.14159
CIRCUM = 2 * PI * RADIUS
```

should be written

```
DATA POVERZ/1.570796
CIRCUM = POVERZ * RADIUS
```

to save unnecessary multiplication of two constants.

5. DON'T CALL LIBRARY ROUTINES UNNECESSARILY

Using some standard FORTRAN capabilities sometimes causes the loading and execution of library routines which take time and space. For example, the innocuous

```
ENDFILE LU
```

causes the routine .TAPE to be loaded and add 10 words to the size of the program. A faster and smaller version is

```
CALL EXEC (3, LU + 400B)
```

which does the same thing. Similarly, at the end of your program, the statement

```
STOP
```

adds 252 words for the privilege of printing the words

```
PROGX:STOP 0000
```

when PROGX terminates.

6. KNOW HOW TO USE ARRAYS

The use of arrays involves a lot of extra calculation and code. To calculate an element of X(I) of a single dimensioned array takes 4 more words. Similarly, double and triple dimension arrays X(I,J) and X(I,J,K) take 8 and 10 words respectively. Therefore, the following subrules apply:

A. Equivalence array elements where possible. Using the following code

```
Y = X(1)**2 + X(2)**2 + X(3)**2
```

takes 15 more words than

```
EQUIVALENCE (X(19,X1), (X(2),X2), (X(3),X3)
Y = X1**2 + X2**2 + X3**2
```

B. Use single dimension arrays instead of multi-dimension.

For example, zeroing an array of 20 by 20 with

```
DIMENSION A(20,20)
DO 10 I = 1,20
DO 10 J = 1,20
10 A(I,J) = 0.0
```

takes 4400 more executing instructions than

```
DIMENSION A(400)
DO 10 I = 1,400
10 A(I) = 0.0
```

even though it is only 15 words longer.

C. Calculate an array element as few times as possible. The equations

```
X(I) = A(I)/B(I)
Y(I) = A(I)/C + W
Z(I) = R/2. + A(I)
```

can best be written

```
AI = A(I)
X(I) = AI/B(I)
Y(I) = AI/C + W
Z(I) = R/2. + AI
```

for a savings of 19 instructions for each time through a loop.

D. Know what the use of an array name without subscript means. In an I/O statement such as

```
WRITE (LU,10) ARRAY
```

the whole array will be written. But in any other statement such as

```
ARRAY = X
```

this is equivalent to writing

```
ARRAY(1) = X
```

which means that 4 words can be saved in any non-I/O reference to ARRAY(1) without the need for an EQUIVALENCE statement described above.

7. AVOID FORMATTER LIKE THE PLAGUE!

If anything takes up code in a program, it is the formatter. A single read/write statement loads the formatter with the program. This can add a minimum of 5130 words to the program size. If at all possible, use EXEC/REIO calls and system library routines KCVT, CNUMO, and CNUMO to do I/O.

For I/O with no variables, you can use

```
        CALL REIO (2,LU, 11H ENTER DATA ,-11)
```

for

```
        WRITE (LU 10)
        10 FORMAT (" ENTER DATA")
```

or

```
        CALL REIO (1,LU + 400B,IBUF,3)
```

for

```
        READ (LU,100) IBUF
        10 FORMAT (3A2)
```

If you are using simple integers, replace

```
           WRITE (LU,10) LUOUT
        10 FORMAT ("CURRENT LU IS",I6)
           READ (LU,11) LUOUT
        11 FORMAT (I6)
```

with

```
        CALL OUT (CURRENT LU IS       ", LUOUT,10,LU)
        CALL EXEC (1,LU + 400B,IBUF,-6)
        CALL ABREG (IA,IB)
        CALL PARSE (IBUF,IVAL,N,LU)
        LUOUT = IRBUF(2)
              ,
              ,
        END
```

If you are doing binary I/O

```
        READ (LU) IBUF
```

or

```
        WRITE (LU) IBUF
```

a

```
        CALL EXEC (1,LU+100B,IBUF,N)
```

or

```
        CALL EXEC (2,LU+100B,IBUF,N)
```

is far superior.

For complicated I/O (tabbing, real, double precision, multiple integer) formatter may be the only logical choice, but if speed of a program is critical, consider using class I/O to pass the unformatted buffer to another program which formats and outputs it.

While it is not possible for even the most efficient FORTRAN programmer to code as efficiently as in Assembly, the use of efficient coding techniques can be the difference between a program that is short, efficient and fast, or one that has to resort to slower, less efficient techniques such as EMA or segmenting. If you are not too sure what code your program is generating, or what extra and possibly unnecessary routines your program is causing a load of, use the FORTRAN M (Mixed) listing and the LOADR list map to assist you.

BEWARE OF OLD FORTRAN CODE

Kent Ferson/HP Data Systems Division

FORTRAN programs compiled with the 24177 FORTRAN compiler (now obsolete) generated unnormalized floating point numbers in the object code. The F-series floating point hardware always expects a normalized floating point number, and will therefore produce garbage results. The M-series and E-series computer will work correctly.

There are two work-arounds for this problem. The first is to recompile the source program using the RTE FORTRAN compiler. The current compilers always generate normalized floating point numbers. If the source is not available, the user can explicitly search the relocatable libraries ($MLIB1 AND $MLIB2) during load time. This second solution forces the software routines to be used instead of the floating point hardware.

PASCAL/1000 PROGRAMMING COURSE

Shauna Uher/HP Data Systems Division

The Pascal/1000 programming course is now being offered at HP Regional Training Centers. This course teaches the concepts of structured programming in Pascal/1000 in the RTE-IVB operating system environment. Completion of the RTE-IVB Session Monitor User's Course is a pre-requisite. No previous Pascal programming experience is required.

The course is divided into ten modules which are covered in five days.

MODULE	TOPIC
1	Introduction, Structured programming, Statements
2	Type declarations, User-defined types
3	Structured types, Data representation
4	Procedures, Functions
5	Introduction to files, file manipulation procedures
6	Sequential and direct access files, File I/O procedures
7	Pointers, Dynamic variables
8	Segments, Subprograms
9	Pascal/1000 and System library routines, FMP, EXEC, FORTRAN routines
10	Discussion Topics

Each of these modules is supplemented by labs which consist of worksheet questions and programming assignments.

The student workbook contains a copy of the overhead slides and a section of student text which describes the slide material. The students also receive the book Programming in Pascal/1000 by Peter Grogono.

Contact your local HP sales office for enrollment and course availability information.

JOIN AN HP 1000 USER GROUP!

Here are the groups that we know of as of September 1980. (If your group is missing, send the Communicator/1000 editor all of the appropriate information, and we'll update our list.)

NORTH AMERICAN HP 1000 USER GROUPS

Area	User Group Contact
Boston	LEXUS P.O. Box 1000 Norwood, Mass. 02062
Chicago	Jim McCarthy Travenol Labs 1 Baxter Parkway Mailstop 1S-NK-A Deerfield, Illinois 60015
New Mexico/El Paso	Guy Gallaway Dynalectron Corporation Radar Backscatter Division P.O. Drawer O Holloman AFB, NM 88330
New York/New Jersey	Paul Miller Corp. Computer Systems 675 Line Road Aberdeen, N.J. 07746 (201) 583-4422
Philadelphia	Dr. Barry Perlman RCA Laboratories P.O. Box 432 Princeton, N.J. 08540
Pittsburgh	Eric Belmont Alliance Research Ctr. 1562 Beeson St. Alliance, Ohio 44601 (216) 821-9110 X417
San Diego	Jim Metts Hewlett-Packard Co. P.O. Box 23333 San Diego, CA 92123
Toronto	Nancy Swartz Grant Hallman Associates 43 Eglinton Av. East Suite 902 Toronto M4P1A2

NORTH AMERICAN HP 1000 USER GROUPS (CONTINUED)

Area	User Group Contact
Washington/Baltimore	Paul Taltavull Hewlett-Packard Co. 2 Choke Cherry Rd. Rockville, MD. 20850
General Electric Co. (GE employees only)	Stu Troop Special Purpose Computer Ctr. General Electric Co. 1285 Boston Ave. Bridgeport, Conn. 06602

OVERSEAS HP 1000 USER GROUPS

London	Dave Thombs (Vice-Chairman) MQAD, Royal Arsenal East WOOLICH, London SE18 England
Amsterdam	Mr. Van Putten Institute of Public Health Anthony Van Leeuwenhoeklaan 9 Postbus 1 3720 BA Bilthoven The Netherlands
South Africa	Andrew Penny Hewlett-Packard South Africa Pty. private bag Wendywood Sandton, 2144 South Africa
Belgium	E. van Ocken University of Antwerp (RUCA) Groenenborgerlaan 171 2020 Antwerp Belgium
France	Jean-Louis Rigot Technocatome TA/DE/SET Cadarache BP.1 13115 Saint Paul les Durance France
Germany	Hermann Keil Vorwerk+Co Elektrowerke Abt. TQPS Rauental 38-40 D-5600 Wuppertal Germany

Although every effort is made to ensure the accuracy of the data presented in the **Communicator,** Hewlett-Packard cannot assume liability for the information contained herein.

Prices quoted apply only in U.S.A. If outside the U.S., contact your local sales and service office for prices in your country.

Printed in U.S.A. 9/80

Part No. 5951-6111