HEWLETT hp PACKARD

computer systems COMMUNICATOR

3000

Reprints

Issue No. 18

Computer Museum

TABLE OF CONTENTS

EDITOR'S NOTE

Since June of 1975 when the COMMUNICATOR was introduced as part of Hewlett-Packard's support service, eighteen issues of the magazine have been published. During the same period, the number of subscriptions to the magazine has greatly increased. For example, while several hundred copies of the first COMMUNICATOR were printed, 3500 copies of issue #16 were printed.

As the number of subscribers has grown, so also has the frequency of requests for copies of the earlier issues. This COMMUNICATOR has been prepared to make available some of the more important information contained in previous issues. All of the material presented here has been previously published in the COMMUNICATOR and, other than Len Croley's article dealing with memory dump procedures, all of the reprints are from issues prior to #13. For your reference, the issue and page number for the original printing of each article have been included in the Table of Contents.

Articles were selected for reprinting because of their usefulness and in consideration of the continued interest shown in them by subscribers. Each article has been reviewed for accuracy, with respect to changes in the HP 3000 environment since the article was originally published, and for completeness. The result is a document which we hope you will find informative at first reading, and useful later as a reference.

A number of people, other than the authors, helped update the articles. They deserve mention here:

Ed Canfield	Wendell Henry
Bob Crum	John Pavone
Robert Day	Bill O'Shaughnessy
Jim Francis	Steve Zinc

Editor
Computer Systems - COMMUNICATOR 3000
HP General Systems Division
5303 Stevens Creek Boulevard
Santa Clara, CA 95051

Address your subscription and distribution correspondence to:

Subscription Supervisor
Computer Systems - COMMUNICATOR 3000
HP Computer Services Division
P.O. Box 61809
Sunnyvale, CA 94088

COMPUTING DIRECTORY SIZE

Bill O'Shaughnessy
General Systems Division

The System Disc Directory currently has the following usable maxima: 630 accounts; 95 groups per account; 200 users per account; and 1385 files per group. The following equation should be used to determine the approximate number of sectors needed for the directory:

SECTORS = 6+6*A +(05.4*G)+(.15*U)+(.05*F)+(.5*VS)+(.5*VC)

Where:	A =	TOTAL NUMBER OF	ACCOUNTS	IN SYSTEM	
	G =	"	GROUPS	"	
	U =	"	USERS	"	
	F =	"	FILES	"	DOMAIN.
	VS =	"	VOLUME SETS	"	
	VC =	"	VOLUME CLASSES	"	

(Note all quantities in parentheses should be rounded up to the next whole number.)

The number yielded by the above equation is the approximate maximum number of sectors required. This formula will most closely approximate the number of sectors required by the directory at initial configuration or after a RELOAD. Normal use of the system (the creation/purging of accounts, files, and so forth) may cause the directory to become fragmented and, consequently, to use additional disc space.

*This equation replaces the one found in Appendix E of the HP 3000 System Manager/System Supervisor manual.

TIPS ON USING EDIT/3000

Dick Sleght
General Systems Division

1) When you sign on to the HP 3000 with a terminal connected to a port that has a smaller configured record width than your terminal, you can use:

   o The formal designator "EDITOUT" to change the output size

   o "EDITIN" to change the input size.

   The example below illustrates the use of an 80-character device on a configured 72-character port. Remember that the FILE command must go before the EDITOR command.

```
  :FILE EDITOUT=$STDLIST;REC=-80
  :FILE EDITIN=$STDINX;REC=-80
  :EDITOR

HP32201A.07.02 EDIT/3000 WED, SEPT 20, 1978, 3:04 PM
/SET LENGTH=80,RIGHT=80
/ADD

  1
  1234567890123456789012345678901234567890123456789012345678
  01234567890
  2          2          3          4          5          6
             7          8
```

With a 132-character terminal you can obtain an editor listing formatted the same as the offline option without folding at the configured width. For example, :FILE EDITOUT=$STDLIST;REC-132.

2)
```
:FILE L;DEV=LP
:EDITOR *L
HP32201A.07.02 EDIT/3000 WED, SEPT 20, 1978, 3:18 PM
/SET FRONT, FROM=10, DELTA=10,SHORT; ADD
  10      TO ADD LINE NUMBERS TO
  20      THE FRONT FOR SEQUENCE
  30      NUMBERS OR BASIC LINE
  40      NUMBERS...
  50      SET FRONT
  60      THEN KEEP IN A FILE.
```

```
       70      TEXT THAT FILE UNNUMBERED.
       80      CHANGE THE UNNEEDED COLUMNS
       90      TO BLANK OR NULL
      100      //
  ...
/KEEP X;SET FROM=1, DELTA=1,REAR;TEXT X,UNNUMBERED
/LIST FIRST
        1      00010000TO ADD LINE NUMBERS TO
/CHANGEQ 1/3 TO "" IN ALL
/LIST FIRST
        1      10000TO ADD LINE NUMBERS TO
/CHANGEQ 3/5 TO " " IN ALL
/LIST ALL
        1      10 TO ADD LINE NUMBERS TO
        2      20 THE FRONT FOR SEQUENCE
        3      30 NUMBERS OR BASIC LINE
        4      40 NUMBERS ...
        5      50 SET FRONT
        6      60 THEN KEEP IN A FILE.
        7      70 TEXT THAT FILE UNNUMBERED.
        8      80 CHANGE THE UNNEEDED COLUMNS
        9      90 TO BLANK OR NULL
/KEEP ABC,UNNUMBERED

/EXIT

:END OF SUBSYSTEM
```

3) Question! How do I list the contents of Z?

Answer! Use Q #Z::#

```
:EDITOR

HP32201A.07.02 EDIT/3000 WED, SEP 20, 1978, 4:26 PM
/Z::=
ENTER Z::=
Z CAN CONTAIN ANY "TEXT", CHARACTERS, OR COMMANDS
/Q#Z::#
Z CAN CONTAIN ANY "TEXT", CHARACTERS, OR COMMANDS
```

Remember any special character except -+/().,;´& can be used to delimit strings.

If Z contains the delimiter the display will only be up to the delimiter.

```
/Q"Z::"
Z CAN CONTAIN ANY
/E

END OF SUBSYSTEM
```

RETRIEVING EDITOR WORK FILES AFTER A SYSTEM CRASH

Madeline Lombaerde
General Systems Division

As most 3000 users know by now, the EDIT/3000 subsystem saves the contents of the user's work area in a file generally named Kdddtttt where ddd is the day of the year and tttt is the time. When the system crashes or the user aborts the EDIT/3000 subsystem, this "k" file is saved in an attempt to preserve the user's work.

However, the various types of system crashes possible make it difficult to predict the exact state of the work file. Forward and backward pointers are stored in the file and if a block was not yet updated when the crash occurred, the pointers may be invalid.

There are certain techniques that may help you recover all or most of your work files. Follow this sequence of steps after you have logged on to the system.

a. Identify the workfile with a :LISTF,2 command.

b. Enter the EDITOR.

c. Activate the file as a workfile using the TEXT command.

d. Assure yourself that the workfile belongs to you and that its linkages are intact in both the forward and backward directions. LIST ALL can be used to check the forward pointers. To check the backward pointers, SET TIME equal to the number of records in the file then use WHILE FLAG;LIST*-2.

e. If the forward pointers are OK, KEEP the file (do not use KEEPQ).

f. TEXT your workfile into the EDITOR; it should be in good order.

Suppose, however, at the time of the crash, the last block (containing pointer information) was not written out. The LIST ALL fails near the end of the file. One solution might be to do a partial KEEP: that is, KEEP with a range that ends just before the line that couldn't be listed. Then TEXT the file in and fix the last lines. Another possibility (providing it's not essential to preserve your line numbers) would be to HOLD the lines that are OK, do a DELETE ALL (this is important), and then ADD from the HOLD file. This will set up a new work file and the lines will be renumbered.

In the case where backward pointers are correct but the forward pointers are not, do a FIND LAST (if possible), or whatever you can do to get to the end of the file. Then set TIME equal to the number of records in the file and use the following:

```
/W
/FQ*
/BEGIN
/HQ*,APPEND
/FQ*-1
/END
```

This will store the records in the HOLD file in reverse order. Then DELETEQ ALL and ADD from the HOLD file. The file will be in reverse order and repeating the process will restore it to its original form.

When there's a "hole" in the file, that is, a few lines in the middle of the work file are not accessible, a partial keep (i.e., KEEP using a range) of the beginning part and then of the end part can help save most of the file. Similarly, multiple KEEP's using ranges can help get around multiple "holes." Do a TEXT of the first part kept and then JOIN the other part(s). A key point to keep in mind is that an ADD after a DELETE ALL will set up a new work file; a TEXT command always sets up a new work file.

MEMORY DUMP PROCEDURES FOR SERIES II (*)

Len Croley
HP General Systems Division

1. Use the following standard procedures to take a dump:

   A. Put a magnetic tape on DRT 6 UNIT 0 with a write ring.

   B. Press both the ENABLE and DUMP switches at the same time.

   C. Check the code which appears in the CIR register on the front panel. (Refer to Section 5 of the Console Operator's Guide, part no. 30000-90013). If the correct code does not appear in the CIR register, go to step 2.

   D. If step 2 has been performed at least once, and if the dump just completed is good (CIR code is correct), then go to step 3.

   E. If the dump worked the first time through, the tape can now be handled by DPAN2 and the stream file DUMPJOB. Go to 3.D.

2. Accomplish the following procedures (Note: Do not rewind the tape):

   A. Put 006606 in the switch register.

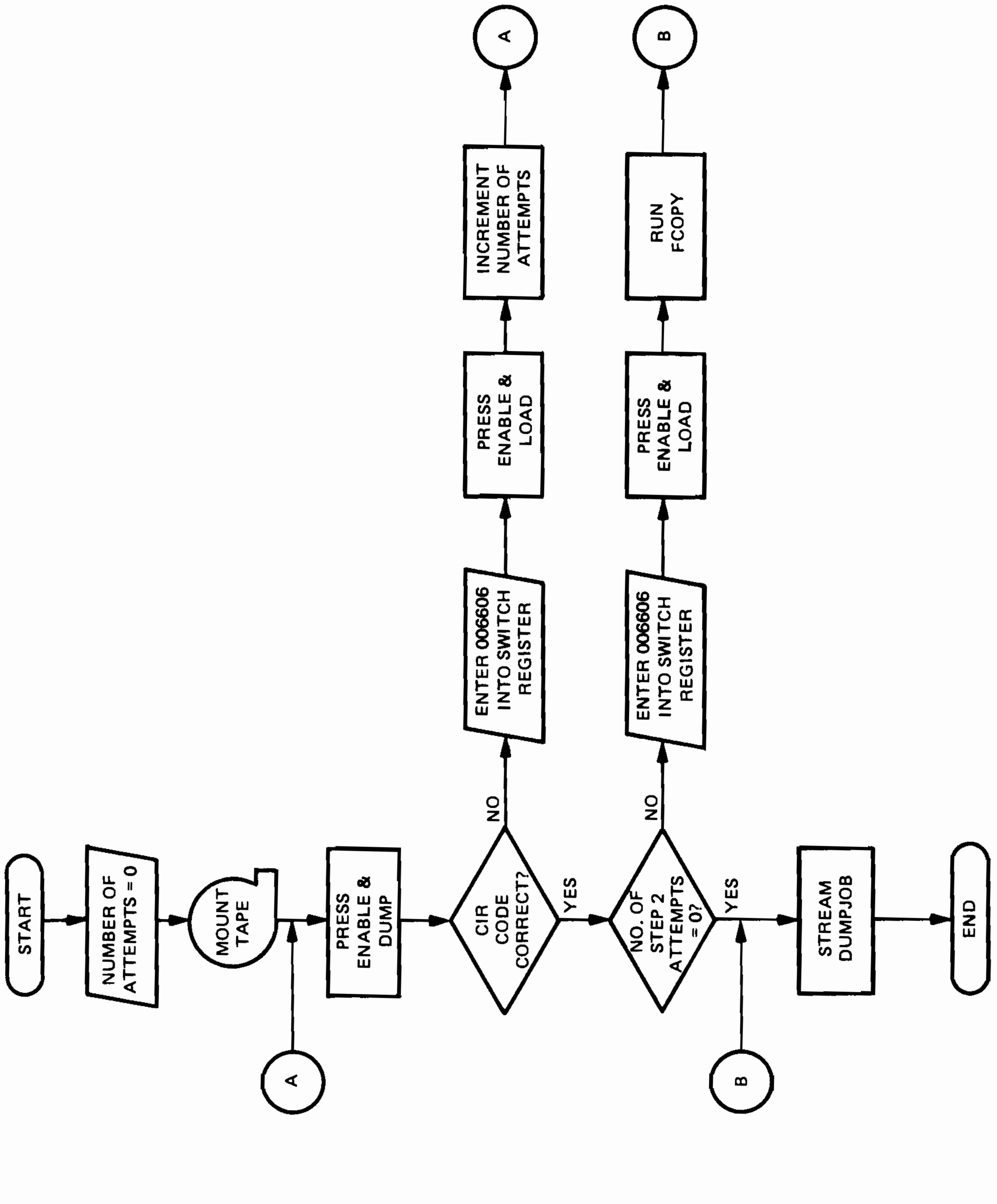
   B. Push the ENABLE and LOAD switches. (The tape will move only a small amount. This writes an end-of-file mark on the tape.)

   C. If you accidentally press ENABLE and DUMP instead of ENABLE and LOAD, rewind the tape or mount another tape and start over at step 1.B.

   D. Make a note each time this sequence (i.e., step 2) is completed.

   E. Perform steps 1.B and 1.C again.

(*) Differences in microcode should make these procedures unnecessary on Series I and III.


START
NUMBER OF ATTEMPTS = 0
MOUNT TAPE
A
PRESS ENABLE & DUMP
CIR CODE CORRECT?
NO
YES
ENTER 006606 INTO SWITCH REGISTER
PRESS ENABLE & LOAD
INCREMENT NUMBER OF ATTEMPTS
A
NO. OF STEP 2 ATTEMPTS = 0?
NO
YES
ENTER 006606 INTO SWITCH REGISTER
PRESS ENABLE & LOAD
RUN FCOPY
B
B
STREAM DUMPJOB
END

3. Accomplish the following steps if step 2 has been performed at least once:

   A. Enter 006606 in the switch register and simultaneously press the ENABLE and LOAD switches. (This writes a final End-Of-File on the tape. The EOF is not required by DPAN2 but will be needed for the following steps.)

   B. Bring up the system.

   C. Log onto the system and issue the following commands:

```
1) :FILE DUMPTAPE;DEV=TAPE;REC=4096,1,F,BINARY
2) :BUILD MDUMP;REC=4096,1,F,BINARY;DISC=100,1,1
3) :RUN FCOPY.PUB.SYS
4) >FROM=*DUMPTAPE;TO=MDUMP;SUBSET;SKIPEOF=(Insert the
   number of times you performed step 2.)
5) >EXIT
```

(NOTE: The formaldesignator of the dump source in DPAN2 is MDUMP. If the dump source specified by the user is other than MDUMP, the file command

```
:FILE MDUMP=filename
```

must be entered before DPAN2 can be run.)

   D. Bring up the system and stream DUMPJOB.PUB.SYS, fill out the Problem Report, and submit this documentation to your Hewlett-Packard service representative.

   E. Purge MDUMP.

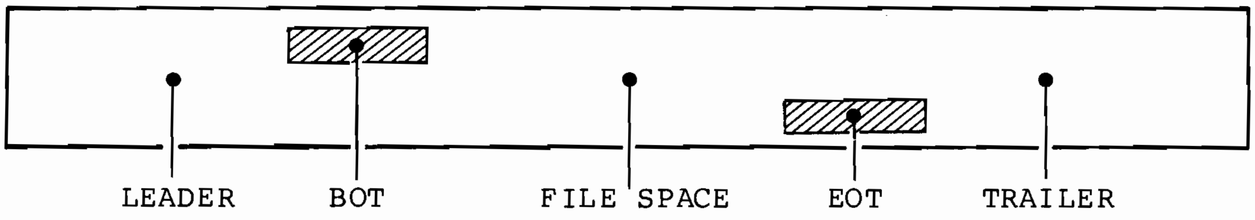
MAGNETIC TAPE CONSIDERATIONS

Denis Winn
General Systems Division

Every standard reel of magnetic tape designed for digital computer use has two reflective markers located on the back side of the tape (opposite the recording surface). One of these marks is located behind the tape leader at the beginning of tape (BOT) position, and the other is located in front of the tape trailer at the end of tape (EOT) position.

These markers are sensed by the tape drive itself and their position on the tape (left or right side) determines whether they indicate the start or end of tape positions. (See below.)

MAGNETIC TAPE



As far as the magnetic tape hardware and software are concerned, the BOT marker is much more significant than the EOT marker. BOT signals the start of recorded information, but EOT simply indicates that the remaining tape supply is running low and the program writing the tape should bring the operation to an orderly conclusion.

The difference in treatment of these two physical tape markers is reflected by the MPE file system intrinsics when the file being read, written, or controlled is a magnetic tape device file. The following paragraphs discuss the characteristics of each appropriate intrinsic.

FWRITE

When a user program attempts to write over or beyond the physical EOT tape marker, the FWRITE intrinsic returns an error condition code (CCL). The actual data has been written to the tape, and a call to FCHECK will reveal a file error indicating END OF TAPE. All writes to the tape after the EOT tape marker has been crossed will transfer the data successfully but will return a CCL condition code until the tape winds off the reel or until a backspace operation (rewind, backspace file) causes the EOT marker to pass the sensor in the reverse direction.

FREAD

A user program can read data written over an EOT marker and beyond the marker into the tape trailer. The intrinsic returns no error condition code (CCL or CCG) and does not initiate a file system error code when the EOT marker is encountered.

FSPACE

A user program can space records over or beyond the EOT marker without receiving an error condition code (CCL or CCG) or a file system error. The intrinsic will, however, return a CCG condition code when a logical file mark is encountered. If the user program attempts to backspace records over the BOT marker, the intrinsic returns a CCG condition code and remains positioned on the BOT marker.

FCONTROL (WRITE EOF)

If a user program writes a logical end of file (EOF) mark on magnetic tape over the reflective EOT marker, or in the tape trailer after the marker, the FCONTROL intrinsic returns an error condition code (CCL) and sets a file system error to indicate END OF TAPE. The file mark is actually written to the tape.

FCONTROL (FORWARD SPACE FILE MARK)

A user program which spaces forward to logical tape file marks (EOFs) with the FCONTROL intrinsic cannot detect passing the physical EOT marker. No special condition code is returned.

FCONTROL (BACKWARD SPACE TO FILE MARK)

The EOT reflective marker is not detected by FCONTROL during backspace file (EOF) operations. If the intrinsic discovers a BOT marker before it finds a logical EOF, it returns a condition code of CCE and treats the BOT as if it were a logical EOF. Subsequent backspace file operations requested when the file is at BOT are treated as errors and return a CCL condition code and set a file system error to indicate INVALID OPERATION.

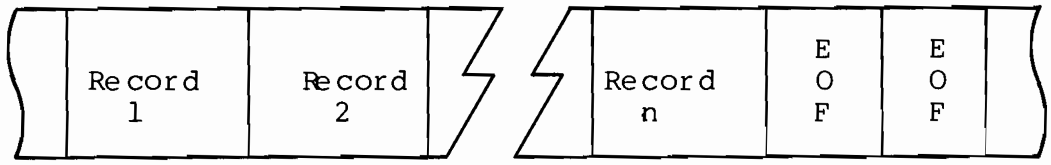
In summary, only those intrinsics which cause the magnetic tape to write information are capable of sensing the physical EOT marker. If a program designed to read a magnetic tape needed to detect the EOT marker, it could be done by using the FCONTROL intrinsic to read the physical status of the tape drive itself. When the drive passes the EOT marker and is moving in the forward direction, tape status bit 5 (%2000) is set and remains on until the drive detects the EOT marker during a rewind or backspace operation. Under normal circumstances, however, it is not necessary to check for EOT during read operations. The responsibility for detecting end of tape and concluding tape operations in an orderly manner belongs to the program which originally created (wrote) the tape.

A program which needed to create a mutli-volume (multiple reel) tape file would normally write tape records until the status returned from FWRITE indicated an EOT condition. Writing could be continued in a limited manner to reach a logical point at which to break the file. Then several file marks and a trailing tape label would typically be added, the tape rewound, another reel mounted, and the data transfer continued. The program designed to read such a multi-volume file must expect to find and check for the EOF and label sequence written by the tape´s creator. Since the logical end of the tape may be located past the physical EOT marker, the format and conventions used to create the tape are of more importance than determining the location of the EOT.

END-OF-FILE MARKS ON MAGNETIC TAPE

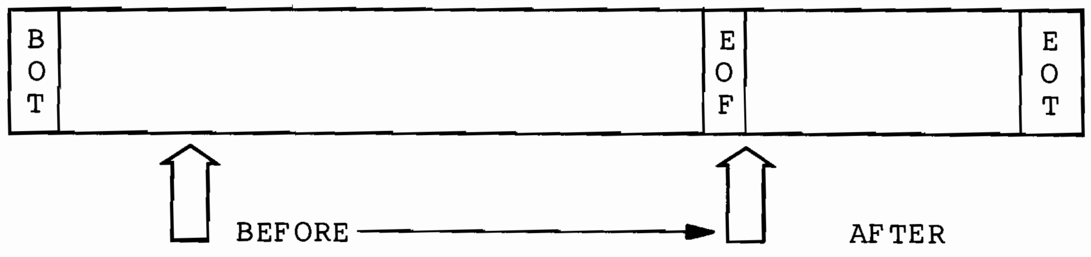
An FWRITE to magnetic tape, followed by any intrinsic call which reverses tape motion (for example, backspace a record, backspace a file, or rewind) causes the file system to write an EOF mark before initiating the reverse motion.

For example, if a user program has just written several data records to magnetic tape, writes a file mark, rewinds the tape, and closes the file, the tape file will be terminated by two file marks (EOF). The first of these was requested by the user by calling FCONTROL to write an EOF, and the second was provided by the system because the direction of tape motion had been reversed after a write (rewind). See below.

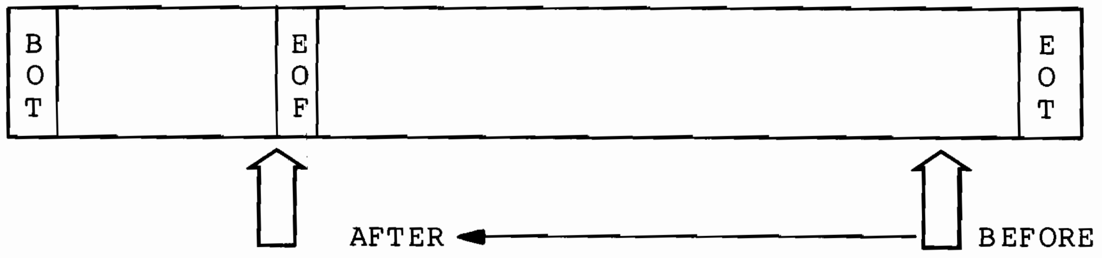


SPACING FILE MARKS

When you space forward to a tape mark (EOF), the tape recording heads have just read the EOF and are positioned beyond it, as follows:



When you space backward to a tape mark (EOF), the mark is recognized as the tape travels in the reverse direction. The tape heads are left positioned just in front of the EOF that was read, as follows:



Note: BOT (beginning of tape) and EOT (end of tape) correspond to the reflective markers on the reel of magnetic tape.

When FREAD has found a logical file mark and returned a condition code of CCG, an EOF mark has been read and the tape heads are positioned immediately following the mark (similar to spacing forward to a tape mark, as described above).

CALCULATING THE APPROXIMATE SIZE OF OUTPUT SPOOFLES

Madeline A. Lombaerde
General Systems Division

An output spoofle contains variable length records; each record has an overhead of 5 words (including carriage control when not embedded in the record). The records are blocked to a maximum of 508 words (4 sectors) per block. Using this information, it is possible to approximate the disc space required by a particular output spoofle.

Example:

6600 lines, each 132 characters long was written into an output spoofle and required 3772 sectors. This number can be calculated by the following method:

1. # words per record = $\lceil 132/2 \rceil + 5 = 71$ words.

2. # records per block = $\left\lfloor \frac{508 \text{ words/block}}{71 \text{ words/record}} \right\rfloor = 7$ records.

3. # blocks = $\left\lceil \frac{6600 \text{ records}}{7 \text{ records/block}} \right\rceil = 943$ blocks.

4. Disc Space = 943 blocks * 4 sectors/block = 3772 sectors.

   Where $\lfloor\quad\rfloor$ means truncate to nearest whole number less than or equal to the value of the expression and $\lceil\quad\rceil$ means round up to nearest whole number greater than or equal to the value of the expression.

Please note that since records normally vary considerably in length, an average record length must be estimated in order to follow these calculations. A certain amount of error will almost always result but at least you'll have a reasonable approximation of the required disc space (*).

It might also be useful to know the approximate number of sectors required for each page of output. The number of sectors required for one page (60 lines) of records of the specified number of characters are shown as follows.

CHARACTERS/LINE	SECTORS PER PAGE (60 LINES)
132	36
118	32
72	20
36	12
18	8

Note that the greater the number of pages to be printed per output spoofle, the more efficiently the Spooler can utilize disc space. In the calculations given above, 3772 sectors were required to print 110 pages, 60 132-character-lines on each page (6600 lines). However, if we multiplied the number of sectors required for one page, the number would be 110 x 36 = 3960 sectors (5% error).

SPOOK UTILITY

The SPOOK utility provides another way of finding out both the number of sectors actually used and the number of printlines produced.

```
:RUN SPOOK.PUB.SYS
SPOOK B00.00 (C) HEWLETT-PACKARD CO., 1976
>SHOW;@
#FILE   #JOB   FNAME  STATE   DEV/CL  PR  CP  RFN    OWNER
#012    #S107   OUT   READY     LP    8   1        USER.ACCT

#FILE  LDEV    LABEL    SECTORS   LINES     TIME
#012    %3   %1026400    1024      169    8:49 8/23/78
>EXIT
:
```

(*) Output spoofles contain variable length records: trailing blanks are truncated.

SPOOLING AND JOB MANAGEMENT NOTES

The information presented in this article is taken from a previously published System Analyst Note (3000.MPE.GENERAL-22, January 10, 1975).

USER FACILITIES

1. JOB SUBMISSION

1.1 Sequential JOBs

Normally, every JOB in a sequence of JOBs is independent of the other. Such JOBs can be submitted and executed in any order. In certain exceptional cases, however, a sequence of JOBs is ordered such that a particular JOB must be completed before the following JOB(s) may be executed, or even submitted. When such a JOB sequence is submitted on a nonspooled device, this ordering is implicitly effected, because every command record is executed when it is physically read. Spooling a job-accepting device can essentially nullify this ordering since: 1) JOBs are admitted (by the spooler) before preceding JOBs have executed; and 2) the concurrency of JOB execution is not necessarily limited to 1.

A JOB contains a grouping of requests which are presumably logically related in the indicated order. Every request pertaining to the JOB´s chore is included; and no function not relating to the JOB is included. That´s why JOBs are treated independently in MPE. When an "ordered" sequence of JOBs is to be submitted, the following can be considered:

1) Why are the JOBs all separate? The applications in which separate ordered JOBs must be submitted are rare. Users are encouraged to combine into one JOB all those functions necessary to implemnt the JOB´s chore. Different JOBs are required only when a separate logon domain is necesary, such as account creation JOBs, or to cause billing to another account, group, or user. [Sometimes, users use separate, ordered JOBs so that the aborting of one step will not cause succeeding steps to be skipped. In these cases, :CONTINUE should be used. The next two suggestions address those applications where an ordered JOB sequence is required.

2) Unspool device. This is obviously the simplest procedure. It makes the JOB sequence ordered because JOB execution is tied to physical reading.

3) "Spawn" JOBs with STREAM. A sequence of two or more disjoint, ordered JOBs can be converted into one JOB, in which every JOB is :STREAMed by its immediate predecessor. For example, the following two sequences of three JOBs are equivalent:

```
:JOB   J1          :JOB   J1
   .                      .
   .                      .
   .                      .
:EOJ               :STREAM ,!
:JOB   J2          !JOB   J2
   .                      .
   .                      .    (leading ":" replaced by "!"
   .                      .
:EOJ               !STREAM ,#
:JOB   J3          #JOB   J3
   .                      .
   .                      .    (leading ":" replaced by "#"
   .                      .
:EOJ               #EOJ   J3
                   !EOD
                   !EOJ   J2
                   :EOD
                   :EOJ   J1
```

Note that the :STREAMed version is properly nested; i.e., the third JOB is :STREAMed by the second, not by the first - disjoint :STREAMed JOBs all originating from the outer (first) JOB would not work. This would not work if STREAM was not enabled, of course. Also, see Note 1.6 for special conventions necessary where nesting :STREAMs.

1.2 :STREAM from Cards

When preparing :STREAM source from cards on the IBM 029 keypunch, remember that certain 029 and ASCII (Hollerith) characters are transposed. In particular, 029´s "!" is not ASCII "!" (it´s ASCII "]"). Place "!" in the :STREAM command (:STREAM,!), or use another character.

1.3 ":STREAMing Nothing"

:STREAM performs a recognition function similar to that done for real devices. For example, the first action is to "flush" (ignore) the stream until a legimate :JOB or :DATA command is found. No error message or job number will appear if no :JOB or :DATA command is found. This will be the case if the prompt character supplied (explicitly or implicitly) by the :STREAM command is different than :STREAM source, as may sometimes be the case, mistakenly, when using a disc file.

1.4 Terminating Batch :STREAMs

Although SESSIONs can terminate :STREAM with break, JOBs must signal end-of-file. The default :STREAM input is $STDIN, so any ":" will terminate the operation, not the STREAMed JOB´s EOJ. For example:

```
:JOB
:STREAM ,!
!JOB
   .
   .
   .
!EOJ
:FILE L . . .
   .
   .
   .
```

will result in the :FILE command being "swallowed" by :STREAM, signaling end-of-file to it. :EOD should be used to terminate :STREAM in JOBs.

1.5 :STREAM :JOB/:DATA Recognition - Read Length

It is possible to extend a terminal input line by using LF. However, because :STREAM reads only the record width of the source file, this should not be used for long :JOB/:DATA commands; use "&" continuation.

1.6 Nested :STREAMs

When one :STREAMed JOB is introduced from within another :STREAMed JOB be sure they use different prompt characters. Consider, for example:

```
:JOB OUTER
   .
   .
   .
:STREAM ,!
!JOB MIDDLE
   .
   .
   .
!STREAM ,#
#JOB INNER
   .
   .
   .
#EOJ
!EOD
!COMMENT THIS IS "MIDDLE"
!EOJ
:EOD
:COMMENT DONE "OUTER´s" STREAM
   .
   .
   .
```

Here "OUTER" initiates one :STREAM JOB ("MIDDLE"); "MIDDLE", in turn initiates "INNER" when "MIDDLE" executes. [This is a method to ensure the desired JOB sequence, see 1.1.1] If "#" hadn´t been used for "INNER", then "OUTER" would wind up initiating both "MIDDLE" and "INNER". Note, also, the proper nesting of EODs.

2. INTERPRETING JOB/SESSION OUTPUT

When JOB $STDLIST to line printers is examined, the following times are noted:

HEADER - Timestamp, T_h
:JOB & WELCOME info - Timestamp, T_j
:EOJ - Elapsed Time, E - Timestamp, T_e
TRAILER - Timestamp, T_t

The following attempts to clear up any confusion regarding the interpretation and relationship of these times:

T_h - the (wall) time that the header is printed;

T_j - the (wall) time that the JOB actually began execution;

E - the time between the job finishing execution and the time that the JOB was introduced, which means that it includes the time that the JOB was waiting to begin processing;

T_e - the (wall) time that the JOB finishes execution;

T_t - the (wall) time that the trailer is printed.

In analyzing these times, keep in mind that they can be off by + one minute, due to rounding. The following relationships can be derived from the definitions above:

$T_h < T_t$

$T_j < T_e <= T_t$

$E >= T_e - T_j$ (E includes waiting time)

$T_h = T_j < T_e = T_t$, when $STDLIST not spooled

$T_j < T_e <= T_h < T_t$, when $STDLIST is spooled

[The equal relationship, above, should be interpreted as "close to".]

2.2 "(INCOMPLETE)" Trailers

The message "(INCOMPLETE)" on a trailer indicates that the spooled output has been interrupted and does not appear in its entirety. This will occur if:

a. The console operator has explicitly interrupted it by command while it was being printed. It can be "deferred" for possible later printing or deleted.

b. A disc I/O error was detected while it was being printed.

c. The system crashed while the output was being created (before FCLOSing the file). The console operator should know the specific reason.

2.3 No Trailer

A trailer will not appear with an output line printer or card punch file if the system crashes during printing of the file; or, if spooled, a spoolee (device) error is detected.

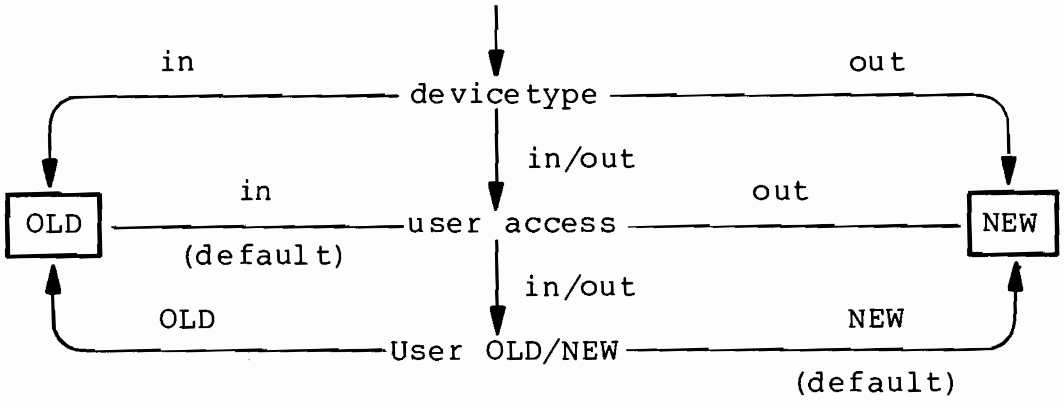
2.4 Card Punch Headers/Trailers

Files produced on spooled card punches are preceded by header cards and followed by trailer cards. These contain only descriptive information about the file which can be read by "interpreting" the cards.

3. DEVICE ALLOCATION

3.1 Allocate Algorithm

The first determination made when allocating a non-sharable device is whether the request is for an "OLD" or "NEW" devicefile:



OLD means to search for a pre-defined input file (e.g., :DATA,$STDIN). If an OLD file cannot be found, the console operator is queried. If a NEW request fails, the user's FOPEN is rejected. The console operator is asked to "locate" all mag tape requests.

Refer to Section IV of the Console Operator's Guide for a discussion regarding the relevant names (job, file) when allocating a :DATA devicefile.

3.2 Allocating :DATA Terminals

:DATA devicefiles can only be allocated as OLD; see Note 3.1. When the target is a card reader, OLD is implied by the device type; i.e., input. But when the :DATA devicefile is a terminal (input/output device), the user specifications become relevant. In particular, he should not request output - only access; and if he's not requesting input access, he must specify OLD. Otherwise, he will get an FOPEN failure.

3.3 Device Assignment for Partially Spooled Classes

When a device class is configured such that some devices on it are spooled and some are not, NEW allocations will prefer available real devices to spooled devices.

SEGMENTATION FOR EFFICIENCY OF SYSTEM-TYPE PROGRAMS

John Page/Madeline Lombaerde
General Systems Division

The purpose of this article is to describe, for the benefit of system programmers, some guidelines for the optimum design of programs for the 3000; in particular, attention will be given to the questions of segmentation.

The 3000 is a process oriented machine, incorporating the separation of code and data, and stack architecture. This permits easy design of re-entrant code. The purpose here is to discuss ways of making a particular process:

a. Run as fast as possible.

b. Have minimum effect on other processes in the system.

PROCESS ENVIRONMENT

When you write a program, it is executed by MPE in the form shown in Figure 1. The process has a single data segment (or "stack") and a variable number of code segments of varying sizes. When you write your program you can control:

a. the size of the stack

b. the number of your code segments

c. the size of each segment

d. which code goes into which segment.

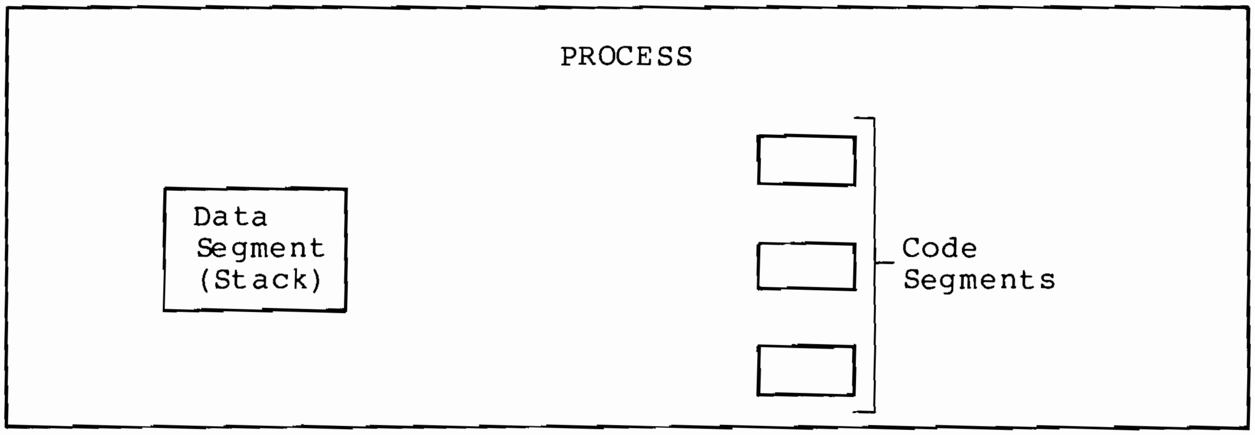


Figure 1.

The preceding diagram is actually a simplification since it does not show the externals referenced by your program (see Figure 2). If for example, your SPL-written program calls FOPEN, then a link will be created from your code to an MPE segment containing the FOPEN intrinsic code. Most of these intrinsics and all the Compiler Library routines are not in memory permanently, thus they are viewed by MPE as code segments identical to your own even though they were not written by you. For programs written in SPL, you are in control of which external procedures are called, since the calls are made explicitly. For other languages, the compiler will implicitly create in your program calls to external routines in order to perform, for example, a Fortran WRITE or a COBOL DISPLAY. The environment of a non-SPL program is harder to control because it requires a knowledge of when the compiler will emit those external calls. We will limit this discussion to those areas over which you have primary control: your own program code and data stack. Given any language, there are some fundamental principles to follow which will decrease the run-time of a process and its impact on system load.

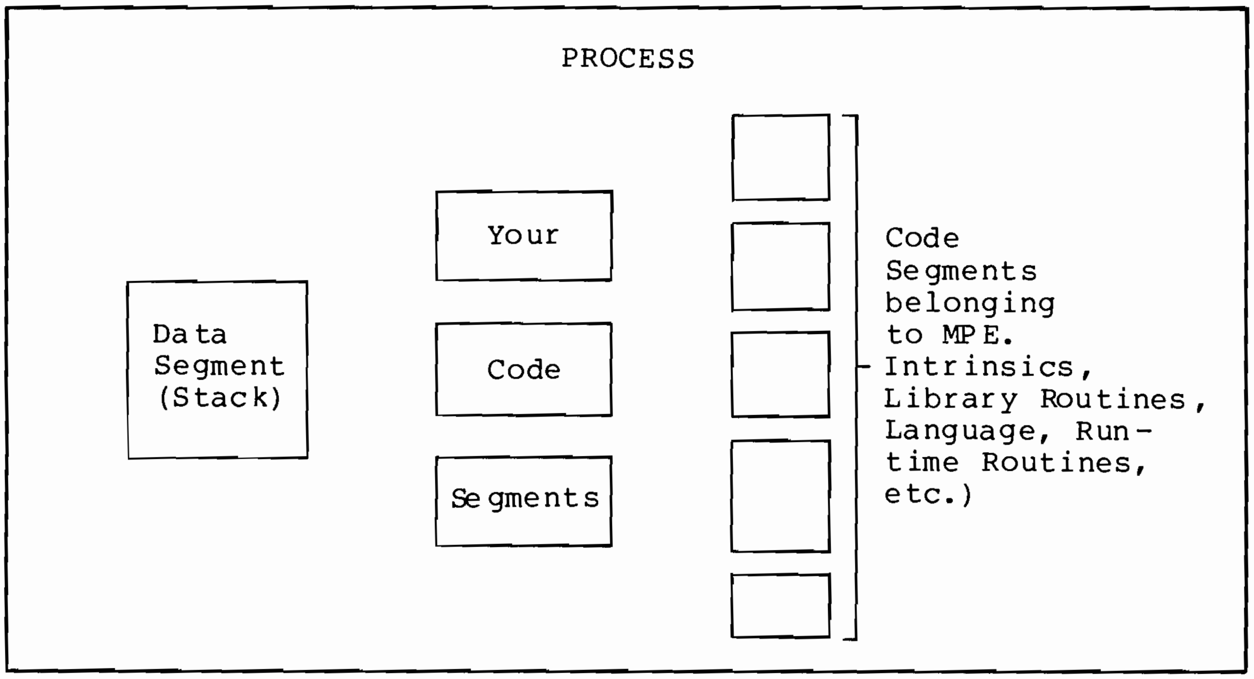

PROCESS
Data Segment (Stack)
Your
Code
Segments
Code Segments belonging to MPE. Intrinsics, Library Routines, Language, Run-time Routines, etc.)

Figure 2.

HOW TO DETERMINE A PROGRAM ENVIRONMENT

When you prepare your program the PMAP option will show the size of each segment, which procedures are in which segment, and the names of externals called by each segment. The MPE Commands and Debug/Stack Dump reference manuals describe the format of the PMAP in detail.

HOW MPE RUNS YOUR PROGRAM

There are two MPE modules concerned here - the dispatcher and the memory management system. The dispatcher is responsible for the allocation of CPU time to all the executing processes. The memory management system has the job of fitting code and data segments into memory as they are required, this operation often requiring the decision of which segment(s) to delete to make space. When your time-slice starts, the stack is made present in memory and control is passed to the program. As the program proceeds, it will call procedures which are not in the current segment. At this point your program is suspended while MPE arranges to make the required segment present. This can take from 20 to 100 milliseconds since a disc access is involved. While this is going on the dispatcher tries to run the process with the next highest priority which is already resident in memory. When the destination segment has been made present, control is passed to the procedure originally called.

The point to note here is that calling a procedure in an absent code segment is a time-consuming job.

HOW DO I TELL IF A SEGMENT WILL BE PRESENT?

You can´t. The memory management system will simply attempt to keep the most popular segments in real memory. The smallest set of segments (both code and data) which must be in real memory for a program to execute efficiently is called the program´s working set. This dynamic set of segments may, and most often does, change continuously during the life of the executing program.

The philosophy of the HP 3000 memory manager is based on the idea that there is an ideal absence frequency for an executing process. If a process gets more than the expected number of absences, the memory manager concludes that the process does not have enough segments in its working set, and proceeds to add the requested (absent) segments to the working set.

However, if a process executes for a long time without absence traps, the memory manager concludes that the working set is too large and real memory is not being used efficiently. The least used segments will be removed from the working set and made available for overlay.

From the user´s point of view, he cannot influence the internal function of the memory manager. The user can, however, design his application with the working set concept in mind. First, he should keep in mind that a total of all working sets active at the same time (i.e., a total of all the commonly used segments, at any given time, for all application programs running concurrently) should be less than or equal to 75% of his linked (i.e., available to user) memory. This restriction on working set size is critical and directly reflects the memory management segment replacement algorithm.

RULES FOR SEGMENTING YOUR PROGRAM

Rule No. 1

Minimize the number of times the program crosses a segment boundary. In other words, stay within a segment for as long as possible. When you leave it, stay out for as long as possible.

DESIGN OF PROGRAMS IS IMPORTANT

Do not leave segmentation to the last minute. As will be shown below, it is possible to write programs that cannot be correctly segmented.

Any procedure or outer block Relocatable Binary Module (RBM) must reside wholly within a segment. Thus if it proves necessary to move a block of code into a separate segment, it will only be possible if the code is a procedure. You cannot take an arbitrary set of instructions and place them into a named segment - the whole RBM must be moved. Therefore, the way you divide your program into procedures is vitally important in the design phase.

CONCEPT OF LOCALITY

The locality of a program is the degree to which control remains in the same general area of code. A high locality means that control remains in the same area for a long period of time. Poor locality means the program branches wildly and rapidly, all over the place. The 3000 needs programs that have good segment locality but does not care about the degree of locality within any given segment. That is to say, it does not want programs that jump from segment to segment continuously but once inside any given segment, it doesn't matter what the locality is like.

If correctly applied, the principle of locality minimizes the number of possible absence traps and minimizes segment switching during execution. Although transferring control between memory resident code segments takes less time than accessing segments on disc, it still requires more execution time (approximately 2.5 times longer) than transferring within the same segment.

FUNCTIONAL vs. TEMPORAL SEGMENTATION

Intuitively, one segments according to the function of the procedures. That is, all the command decoding routines are put together, the command executors are put together, etc. This is wrong. Wrong! Segmentation is a speed-enhancing operation; time, not function, is the critical dimension. Since Rule No. 1 says stay inside a segment for as long as you can, control must pass smoothly.from segment to segment as the program progresses.

As an example, consider a small utility program which dumps a file to the line printer in some special format. Let us suppose that the operator can choose the name of the file and which of three possible formats to use. The program is written with four procedures: A, B, C, and D.

Let us further suppose that each dump routine has a procedure to fetch a record from its file and a procedure to format a print line:

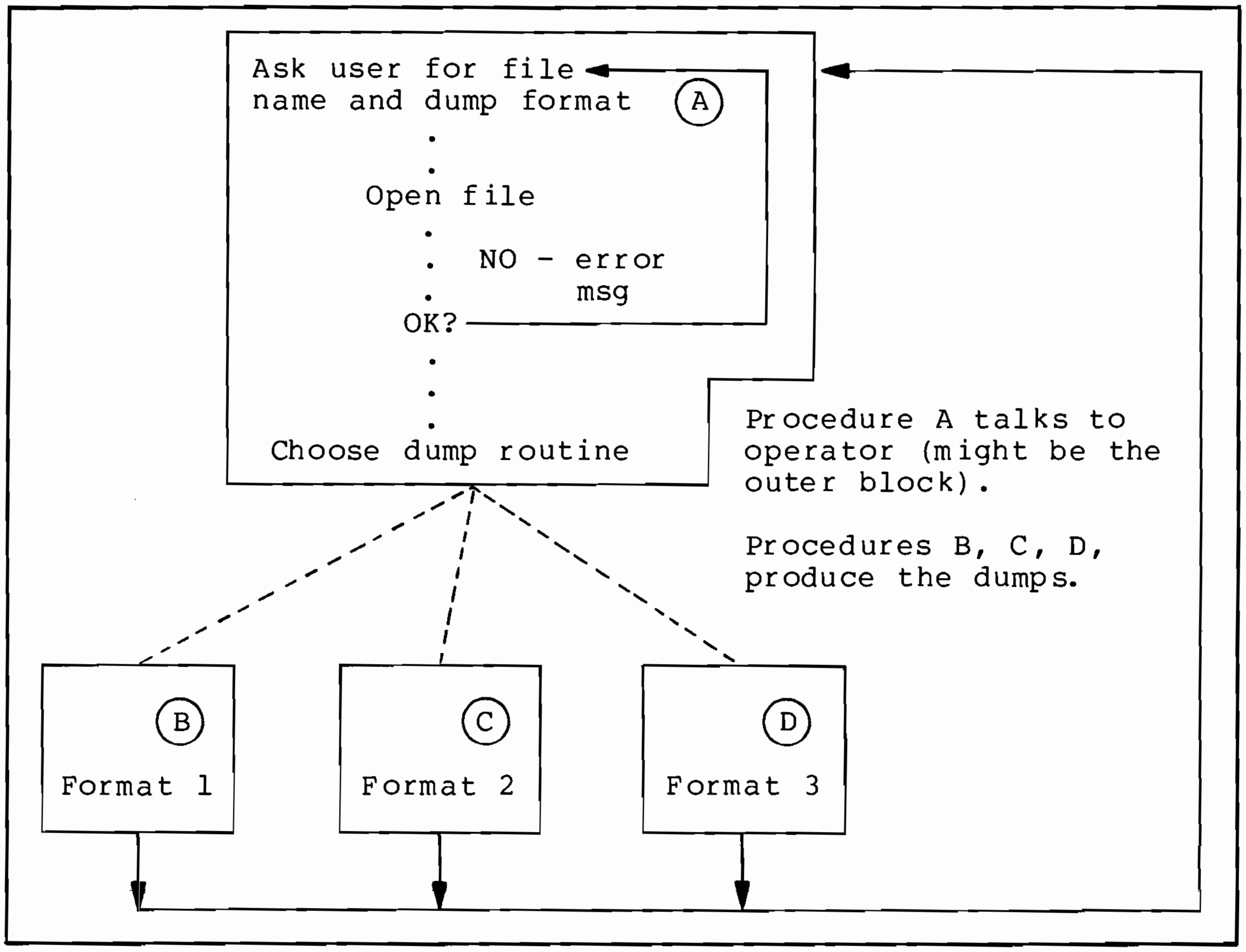

Ask user for file name and dump format
A
Open file
NO - error msg
OK?
Choose dump routine
Procedure A talks to operator (might be the outer block).
Procedures B, C, D, produce the dumps.
B
Format 1
C
Format 2
D
Format 3

Figure 3.

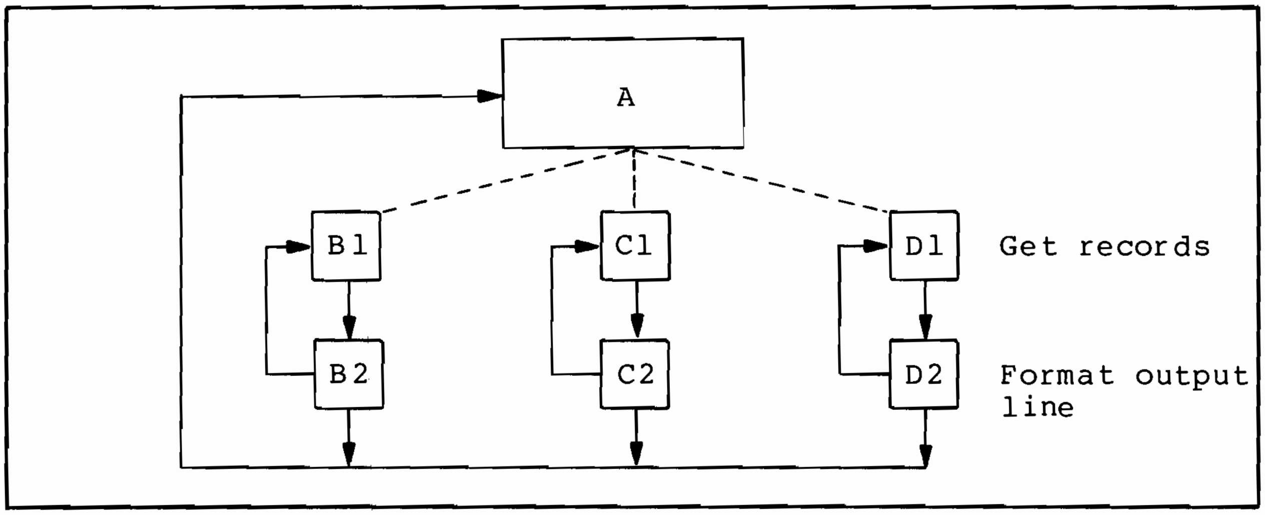


Figure 4.

It would be tempting to put all the formatting routines in one segment, and the record fetching routines in another. This would cause a segment boundary to be crossed twice for every record dumped - perhaps a thousand times. The correct way is to put B1B2 together, C1C2, etc. If A is in its own segment then only three segment boundaries are crossed for a whole dump. In a busy system this simple change could make large differences in the run time of your program.

To sum up, estimate the number of times a segment boundary is crossed in your program and multiply this by 40 milliseconds. This is the time your program could be doing no useful work and other processes will be disrupted.

Rule No. 2

DO NOT BURDEN YOUR WORKING SET WITH INFREQUENTLY USED CODE

Let us suppose that you have arrived at some segmentation scheme using the above rule so that you have good segment locality. The next step is to reduce the size of the ´working set´.

FREQUENCY OF CODE USE

The ´working set´ of segments is the set that consumes most of the CPU time. For example in the program above the working set is the code that executes the main loop such as C1C2. Let us assume that C1C2 are in a segment of their own called CSEG. The system may spend minutes in this segment for a large dump. It is important, therefore, to minimize its size in order to reduce competition for the scarce memory.

To do this, examine the code in the working set and remove any code that executes infrequently. Very often, this applies to code which does error-handling. When your program detects an error, do not handle it in-line. Write an error-message generating procedure and call it with a parameter indicating which message to output. This can be put in a separate segment and thus not clutter up memory while doing normal error-free processing. As another example, suppose that in the program mentioned above, after doing an FWRITE, you check the condition code for end-of-file and, if required, execute a somewhat elaborate sequence to extend the file by building a new one and copying the old into it and then purging the old file. If this condition is likely to occur once in every 500 runs, why hold it in precious memory with the working set? Banish it to some auxiliary segment and let MPE bring it in only when needed. Remember that you can only move this code if it is a procedure.

```
WRONG                                   RIGHT

FWRITE(...);                            FWRITE(...);
IF>THEN                                 IF>THEN EXTEND´FILE;
BEGIN
.
.
<<LENGTHEN FILE>>                       Procedure EXTEND´FILE
.                                       is put in another
.                                       segment
END;
```

SEGMENT SIZES

This is a trade-off. If you have a lot of small segments, then they are easier for the memory manager to place in real memory . However, a scarce resource is being used up in the form of Code Segment Table Extension (CSTX) entries. One entry in the CSTX is needed for every program segment, and the table has a maximum of 63 entries per program executing.

At the opposite end of the spectrum, your program might have a few large segments. While this does minimize segment-boundary crossings, the effect on memory can be devastating for other users. There is no simple answer to the question of optimum segment size. The main idea is to minimize the size of the working set.

Rule No. 3

Keep the principal working set small and make infrequently used segments large.

IF YOUR CODE IS SHARED

If your program is going to be run from multiple terminals then the code segments will automatically be shared by the multiple processes. Each process will have its own stack, of course. If your program design requires data which is never altered, such as error messages, look-up table, etc., then by placing them in the code rather than the stack, only one copy is required for all processes.

WRONG

```
BEGIN
BYTE ARRAY MESSG (0:22):="TOO MANY TIMES ENTERED";    Global
                                                      Declarations
   .
   .
   .
PROCEDURE MESSOUT;   Procedure to print error message
BEGIN
PRINT (MESSG,-23,0);
   .
   .
END;
   .
   .
END.
```

WHY WRONG?   The array MESSG is present in the stack perpetually. Each process running this program carries the message string around in its stack.

RIGHT

```
BEGIN   MESG only exists while MESSOUT executes.  SPL will store
   .    the string in quotes in the code segment - effectively
   .    making it shared.  The stack is now smaller.
   .
   .
PROCEDURE MESSOUT;
BEGIN
BYTE ARRAY MESG(0:22);
MOVE MESG:="TOO MANY VALUES ENTERED";
PRINT (MESG,-23,0);
END;
   .
   .
   .
END.
```

GENMESSAGE

Another way to keep the principal working set small is to use the new message system provided by MPE III through the GENMESSAGE intrinsic. This intrinsic allows you to keep error messages in a disc file, avoiding placing them in either the stack or code segments. The messages can be read from disc whenever needed.

Implementing this idea involves the following steps:

- creating a message catalog using the EDITOR subsystem and MAKECAT program
- opening the catalog in your application program
- including in your application a procedure for selecting and displaying messages from the catalog.

For example:

```
BEGIN
 .
 .
 .
 .
PROCEDURE MESSOUT(MESS'NUM);
   VALUE MESS'NUM;
   INTEGER MESS'NUM;
BEGIN
GENMESSAGE(FILE'NUM,SET'NUM,MESS'NUM)
END;
 .
 .
 .
 .
END.
```

In this example, FILE'NUM is provided for the message catalog at FOPEN, and SET'NUM is a predefined value. The GENMESSAGE intrinsic, using the value in MESS'NUM, will select the proper message from your catalog and print it. Other capabilities of GENMESSAGE are described in the MPE Intrinsics reference manual.

Rule No. 4

In SPL, keep initialized variables, especially arrays, out of the GLOBAL DECLARATIONS.

In Fortran, infrequently used variables and arrays should not be initialized in DATA statements.

SEGMENTATION IN COBOL

Greg Gloss
General Systems Division

COMPILE TIME SEGMENTATION

To use the Segmenter effectively with COBOL programs, you should first understand how the COBOL compiler generates code segments. A COBOL program produces two or more code segments depending on the number of sections in the Procedure Division. The first segment contains one unit which initializes the run-time data area. For the main program, this initialization unit is the Outer Block. The unit name is formed by appending an apostrophe to the name specified in the PROGRAM-ID paragraph.

By using sections with different priority numbers in your program, you can segment your program into several code segments. For COBOL subprograms, the name of the PROGRAM-ID paragraph is used for the unit and segment name of the first such segment. This unit contains the subprogram's primary entry point. In addition, a secondary entry point is present which is used to perform some initialization functions outside of the initialization procedure. The name for this secondary entry point is formed by appending an 'S to the name in the PROGRAM-ID paragraph. All remaining subprogram segments and all main program segments except the outer block derive their names from the section name and priority number of the first section in the segment. In the case of a main program without any sections, the name is formed from the first paragraph name.

If you do not segment your program, it will be put into one segment. The program shown below produces three segments, one for initialization and two for the Procedure Division code.

```
001000   IDENTIFICATION DIVISION
001100   PROGRAM-ID. MAIN.
001200   ENVIRONMENT DIVISION.
001300   DATA DIVISION.
001400   PROCEDURE DIVISION.
001500   FIRST-SEC SECTION.
001600   START-1.
001700     DISPLAY "START OF MAIN PROGRAM."
001800     CALL "SUB."
001900   SECOND-SEC SECTION 02.◄———Priority Number
002000   START-2.
002100     CALL "SUBA."
002200     DISPLAY "END OF MAIN PROGRAM."
002300     STOP RUN.
```

When using the Segmenter to move COBOL program units around, the following restrictions apply:

1. The outer block must be the first program unit presented at PREP time which allocates Secondary DB storage. The base address for the main program data area must be DB+0.

2. The outer block and non-dynamic subprograms cannot be put into an SL.

When studying the following examples, remember that the Segmenter inserts new procedures and new segments at the top of the list. Therefore, you may have to take action to put things in the proper order.

The following items allocate Secondary DB storage and therefore cannot precede the outer block:

1. Procedures from non-dynamic COBOL subprograms which contain the subprogram´s primary entry point.

2. SPL procedures with TRACE, EXTERNAL, or OWN variables.

3. FORTRAN procedures with DATA, COMMON, LABELED COMMON, or TRACE variables.

Dynamic COBOL subprograms and other procedures which do not use global storage may precede the outer block. Segments from non-dynamic COBOL subprograms other than the first segment (i.e., the segment containing the primary entry point) fall into this category.

Consider the USL file, shown below, which was produced by compiling a main program containing two Sections in the Procedure Division.

```
                              Outer Block Designation
MAIN
  MAIN´             75  [OB]  A   C   N
SECONDSEC02´
  SECONDSEC02´      34   P    A   C   N   R
FIRSTSEC00´
  FIRSTSEC00´       36   P    A   C   N   R
```

Now, suppose you have compiled a non-dynamic subprogram into another USL file and want to move it into the master USL file. The first step is to COPY the two subprogram segments into the master USL file using the Segmenter.

```
USL DOCUSL
AUXUSL DOCAUX
COPY SUB                    (Copy first subprogram body segment)
COPY SECONDSUBSEC02´        (Copy second subprogram body segment)
COPY SUB´                   (Copy subprogram initialization)
LISTUSL
```

```
USL FILE DOCUSL.HP32213.SUPPORT

SUB´
   SUB´                153  P    A   C   N   R
SECONDSUBSEC02´
   SECONDSUBSEC02´      51  P    A   C   N   R
SUB
   SUB                  62  P    A   C   N   R◄——Subprogram Primary
   SUB´S                    CP   A   C       R    Entry Point
MAIN
   MAIN´                75  OB   A   C   N
SECONDSEC02´
   SECONDSEC02´         34  P    A   C   N   R
FIRSTSEC00´
   FIRSTSEC00´          36  P    A   C   N   R
```

However, this leaves the USL file in an improper condition since a non-dynamic subprogram primary entry point precedes the Outer Block. There are two ways to correct this situation. The first way is to move the Outer Block to the first segment by using a NEWSEG command as shown below.

```
NEWSEG SUB´,MAIN´
PURGERBM SEGMENT,MAIN
LISTUSL
```

```
USL FILE DOCUSL.HP32213.SUPPORT

SUB´
   MAIN´                75  OB   A   C   N
   SUB´                153  P    A   C   N   R
SECONDSUBSEC02´
   SECONDSUBSEC02´      51  P    A   C   N   R
SUB
   SUB                  62  P    A   C   N   R
   SUB´S                    CP   A   C       R
SECONDSEC02´
   SECONDSEC02´         34  P    A   C   N   R
FIRSTSEC00´
   FIRSTSEC00´          36  P    A   C   N   R
```

The second way is to use the NEWSEG command to create a new segment for the Outer Block as shown in the following example.

```
NEWSEG NEWMAIN,MAIN´
LISTUSL

USL FILE DOCUSL.HP32213.SUPPORT

NEWMAIN
   MAIN´                 75  OB   A   C   N
SUB´
   SUB´                 153  P    A   C   N   R
SECONDSUBSEC02´
   SECONDSUBSEC02´       51  P    A   C   N   R
SUB
   SUB                   62  P    A   C   N   R
   SUB´S                     CP   A   C       R
SECONDSEC02´
   SECONDSEC02´          34  P    A   C   N   R
FIRSTSEC00´
   FIRSTSEC00´           36  P    A   C   N   R
```

The USL file can now be prepared.

If you want to add a dynamic subprogram or a procedure in another language (such as SPL or FORTRAN) which does not use global storage, the COPY command can be used without any further action.

```
COPY SUBA´               (Copy subprogram initialization)
COPY SUBA                (Copy first subprogram body unit)
COPY SECONDDSUBSE02´     (Copy second subprogram body unit)
LISTUSL

USL FILE DOCUSL.HP32213.SUPPORT

SECONDDSUBSE02´
   SECONDDSUBSE02´         51  P   A C N R
SUBA
   SUBA                    52  P   A C N R
   SUBA´S                      CP  A C   R
SUBA´
   SUBA´                  233  P   A C N R
NEWMAIN
   MAIN´                   75  OB  A C N
SUB´
   SUB´                   153  P   A C N R
SECONDSUBSEC02´
   SECONDSUBSEC02´         51  P   A C N R
SUB
   SUB                     62  P   A C N R
   SUB´S                       CP  A C   R
SECONDSEC02´
   SECONDSEC02´            34  P   A C N R
FIRSTSEC00´
   FIRSTSEC00´             36  P   A C N R
```

Now, suppose you want to combine some segments together. You can use the NEWSEG command to combine initialization procedures, main program modules and subprogram compilations.

```
NEWSEG SUB,SUB´
PURGERBM SEGMENT,SUB´
NEWSEG SUBA´,SUBA
PURGERBM SEGMENT,SUBA
LISTUSL

USL FILE DOCUSL.HP32213.SUPPORT

SECONDDSUBSE02´
   SECONDDSUBSE02´       51  P   A C N R
SUBA´
   SUBA                  52  P   A C N R
   SUBA´S                    CP  A C   R
   SUBA´                233  P   A C N R
NEWMAIN
   MAIN´                 75  OB  A C N
SECONDSUBSEC02´
   SECONDSUBSEC02´       51  P   A C N R
SUB
   SUB´                 153  P   A C N R
   SUB                   62  P   A C N R
   SUB´S                     CP  A C   R
SECONDSEC02´
   SECONDSEC02´          34  P   A C N R
FIRSTSEC00´
   FIRSTSEC00´           36  P   A C N R
```

You can also move a main program segment into another segment.

```
NEWSEG SUBA´,FIRSTSEC00´
PURGERBM SEGMENT,FIRSTSEC00´
LISTUSL

USL FILE DOCUSL.HP32213.SUPPORT

SECONDDSUBSE02´
   SECONDDSUBSE02´  51   P    A    C    N    R
SUBA´
   FIRSTSEC00´      36   P    A    C    N    R
   SUBA             52   P    A    C    N    R
   SUBA´S                CP   A    C         R
   SUBA´           233   P    A    C    N    R
NEWMAIN
   MAIN´            75   OB   A    C    N
SECONDSUBSEC02´
   SECONDSUBSEC02´  51   P    A    C    N    R
```

```
SUB
   SUB´               153  P    A   C   N   R
   SUB                 62  P    A   C   N   R
   SUB´S                   CP   A   C       R
SECONDSEC02´
   SECONDSEC02´        34  P    A   C   N   R
```

COBOL code modules (except for the outer block) can be put into an RL.

In summary, the following items can be put into an RL:

COBOL subprograms
Procedures in other languages
Subprogram initialization procedures
Main Program modules.

The following cannot be put into an RL:

Outer Block (Main program initialization).

The following can be put into an SL:

Dynamic subprograms
Subprogram initialization routines
Procedures in other languages without global data.

The following cannot be put into an SL:

Outer Block (Main program initialization)
Non-dynamic COBOL subprograms
Procedures in other languages with global data.

HOW TO BUILD AN INTERFACE BETWEEN COBOL AND ANY MPE INTRINSIC FROM SPL

Terry Von Gease
HP General Systems Division

There are few (if any) MPE Intrinsics callable directly from COBOL. This is due to the way COBOL sends parameters (in the USING clause) and the way the various MPE Intrinsics expect to receive them.

Some MPE Intrinsics are "Option Variable". This means that some of the parameters are optional. When an option variable procedure is called one or more extra words will be passed along with the parameters to indicate which parameters are present. The extra word, or words, are passed, one word for each 16 possible parameters. Bits are set on or off to indicate which parameters are present and which are not. The bits in the extra word, or words, correspond to the actual parameter list on a one for one basis from right to left. When calling an option variable intrinsic it is the responsibility of the calling process to provide this additional information. SPL handles this automatically, however, COBOL will not provide the proper data so any option variable intrinsic cannot be called directly from COBOL.

Also, many of the MPE Intrinsics are "Typed Procedures". A Typed Procedure returns a value (one, two, or four words) that is independent of any passed parameters. This is a value generated by the procedure itself. For example, assume that we have an SPL program that is going to call the MPE Intrinsic BINARY. This intrinisc will convert a byte array containing ASCII numeric characters for a specified length to a one word integer and return this integer as the value of the intrinsic. For this example assume that we have a byte array called BUFFER that contains the character string "1234", we have a one word integer called LENGTH that contains the number 4 (4 characters in BUFFER), and we have another one word integer called RESULT which will be used to contain the result. The actual call to BINARY would look like this:

```
RESULT:=BINARY(BUFFER,LENGTH);
```

Note that the ":=" is SPL for "replaced by", a single "=" means something else.

In this case the value of the integer RESULT is replaced by the binary numeric value of the character string BUFFER. COBOL has no way of coping with this returned value so this MPE Intrinsic is useless to us on that basis alone. (Not to mention what it does to the stack internally).

Another feature of many MPE Intrinsics is the use of the Condition Code to indicate the result of a call to a particular intrinsic. In the preceding BINARY example the condition code would be set to < (less than) if the character string contained one or more non-numeric characters (except a leading minus in this case), a > (greater than) if the result was beyond the capacity of a one word integer (less than -32768 or greater than +32767), or = (equal) if the call functioned properly and the returned value in RESULT is valid. In SPL it is a simple matter to interrogate the condition code after a call, we simply say:

```
IF < THEN ....   (if the condition code is less than)
IF > THEN ....   (if the condition code is greater than)
IF = THEN ....   (if the condition code is equal)
IF <> THEN ...   (if the condition code is not equal)
IF <= THEN ...   (if the condition code is less than or equal)
IF >= THEN ...   (if the condition code is greater than or equal)
```

What follows the THEN statement in each case is code for whatever you want to do if the statement is true.

Since COBOL has no way of interrogating the condition code we would have no way of knowing whether or not the call to BINARY worked properly even if we could call BINARY directly.

Furthermore, if we examine the description of the BINARY intrinsic in the MPE Intrinsics Manual we find that not only is BINARY a procedure that sets the condition code, but also that the character string parameter must be a byte array passed by reference (COBOL passes word arrays by reference, so that's out) and that the length parameter is a one word integer (so far so good) passed by value (so we lose here also).

The difference between a reference and a value parameter is this: When a reference parameter is used the ADDRESS of the parameter is passed; when a value parameter is used the actual value of the parameter is passed. In the case of our call to BINARY, the address of the byte array BUFFER and the binary value 4 are passed. Let's take a look and see what actually happened when we called BINARY:

We know (or we should know) that we have something called a stack for our program. The stack contains all of the data area for our program and is referenced by the STACK or S register (there are other references but S is all we need to consider for this example).

When the SPL compiler encounters the call to BINARY it generates code which does the following:

1. SPL interrogates the MPE Intrinsic file to determine the various characteristics of BINARY.

2. SPL notices that this procedure will return a one word integer value. To accomodate this a zero is loaded onto the stack and the S register is incremented by 1.

3. SPL sees that the first parameter is by reference, so it places a one word address of the first variable specified (BUFFER in this case) on the stack and the S register is incremented by 1.

4. SPL sees that the next parameter is passed by value so it places the value of the specified parameter (LENGTH in this case) on the stack and the S register is incremented by 1.

5. SPL sees that all of the parameters have been taken care of so it generates a PCAL machine instruction for BINARY.

6. PCAL places 4 more words on the stack and increments the S register by 4. These 4 words are referred to as a "Stack Marker" and contain all of the information needed to return to the proper place in the calling program.

7. PCAL does whatever is necessary to make BINARY present in the system and transfers control to it.

8. BINARY does whatever it is that BINARY´s do. One of the things that it will do for sure is place the result value back in the stack in the location of the first word that SPL placed there.

9. BINARY exits. When this happens all of the passed parameters (both of them in this case) and the 4 word stack marker are removed from the stack and the S register is decremented by the proper value (6 in this case). Note that the first location that was for the returned value is NOT removed from the stack. The condition code is set and control returns to the proper place in the calling program.

10. SPL remembers that this procedure will return a one word value so it removes it from the stack and places it in RESULT. The S register is now decremented by 1.

11. We are now back at the start of the next sequential instruction from where we left and the stack is back to what it was before the call to BINARY. The pot´s right.

All of this may or may not make any sense at this time but it does serve to point out what we must do to get at any MPE Intrinsic from COBOL. We must construct an interface to resolve the parameter types, return variables, and the condition code. At this time a few words about COBOL data types may be in order.

1. Any field included in a USING clause MUST be located on a word boundary. This is easily accomplished by defining everything that may be used as a parameter at an 01 or 77 level. If this proves to be annoying, word alignment may be

insured by using the SYNC clause or merely by insuring that there is ALWAYS an EVEN number of bytes between parameters (this method is not recommended).

2. COBOL keeps numeric data in four forms:

   A. The display numeric form of PIC S9(3). All this is a string of ASCII numeric characters.

   B. The edited numeric form of PIC ZZZZZ. This is still a string of ASCII characters.

   C. The COMPUTATIONAL-3 form of PIC S9(3) COMP-3. This, of course, is packed decimal information with 2 digits per byte except for the last byte which contains 1 digit and the sign.

   D. The COMPUTATIONAL form of PIC S9(3) COMP. This is numeric data in integer form. COBOL uses no floating point data, all binary numeric data is kept in one to four word integers. Four or fewer digits are one word; from 4 to 9 digits are two words; and greater than 9 digits are four words.

   Generally only numeric data of type COMPUTATIONAL will be of any use to us for passing numbers. And of the COMPUTATIONAL types only the one and two word integers will be used.

Now let us define and construct an interface so COBOL may call BINARY. The simplest way to do this is to write the calling program in COBOL and then write the interface in SPL.

The portions of the COBOL program doing the calling should look something like this . . .

In the working-storage section:

```
01  BUFFER PIC X(4).
01  LENGTH PIC S9(4) COMP.
01  RESULT PIC S9(4) COMP.
01  COND-CODE PIC S9(4) COMP.
```

In the procedure division (assume that the COBOL program has moved "1234" to BUFFER and 4 to LENGTH):

```
    CALL "CBINARY" USING BUFFER, LENGTH, COND-CODE,
    RESULT.
```

Note that we have called our procedure CBINARY and we have included a couple of extra parameters, as you surely must have guessed, to contain the condition code and the result value.

The SPL interface will have to look something like this . . .

```
1.  $CONTROL SUBPROGRAM
2.  BEGIN
3.  PROCEDURE CBINARY (BUFFER,LENGTH,COND´ CODE,RESULT);
4.  ARRAY BUFFER;
5.  INTEGER LENGTH,COND´ CODE,RESULT;
6.  BEGIN
7.    BYTE ARRAY BBUFFER(*)=BUFFER;
8.    INTRINSIC BINARY;
9.    RESULT:=BINARY(BBUFFER,LENGTH);
10.   IF < THEN COND´ CODE:=-1 ELSE IF > THEN COND´ CODE:=1
        ELSE COND´ CODE:=0;
11. END;
12. END.
```

Note that the line numbers are NOT used for SPL but only included so we may reference each line for discussion . . .

Line 1: This puts the compiler into SUBPROGRAM mode.

Line 2: All SPL programs of any sort must start with a BEGIN statement.

Line 3: This declares the procedure and all of its parameters to the compiler.

Line 4: Each of the parameters included in the procedure declaration (line 3) must now be described. Since COBOL always passes word pointers by reference we declare BUFFER to be a word array.

Line 5: The rest of the parameters are declared as integers since we have described them to COBOL as one word integers. Note that we had to change the name of COND-CODE slightly since SPL does not allow the use of the hyphen. In fact the only special character SPL allows in names is the apostrophe.

Line 6: Like the program itself, each procedure must start with a BEGIN statement.

Line 7: This is a local data declaration and is the heart of the whole interface. We have declared a byte array that shares the same storage location as the word array BUFFER. We call this "Equivalencing" the two arrays. Byte arrays may always be equivalenced to word arrays but never word arrays to byte arrays since there is no certainty of having a byte array begin on a word boundary. We could gain or lose a leading byte.

Line 8: This declares that BINARY can be found in the MPE Intrinsic file.

Line 9: This is the actual call to BINARY. Note that we used the parameters LENGTH and RESULT just as they were passed but we had to use the equivalenced array BBUFFER for the string parameter. When SPL encounters the LENGTH parameter, even though this parameter was passed by reference, it will take the data found at the address that was passed. In other words, SPL will always resolve value parameters whether the value parameter is a literal or a variable. If we knew somehow that there were exactly four characters contained in BUFFER (or BBUFFER since both of these are really references to the same thing) we could have called BINARY thusly:

```
RESULT:=BINARY(BBUFFER,4);
```

In this case the SPL compiler would have immediately placed the binary value four on the stack rather than have gone to the storage location LENGTH and used what it found there. The results would have been identical in either case.

Line 10: This takes care of the condition code. The passed parameter COND-CODE is set to -1 if the condition code was <, +1 if the condition code was >, or 0 if the condition code was =.

Line 11: This terminates the procedure.

Line 12: This terminates the program.

Now that we have written the source for both the COBOL main program and the SPL subprogram we can compile all of this into one USL file and PREP it into a runnable PROGRAM file. To do this always make sure that all of the non-COBOL source is compiled first, then compile the COBOL source. This is because of the way COBOL must link things internally.

What we have done is resolve any required byte arrays by declaring a local byte array that is equivalenced to the passed word array, included an extra parameter from COBOL to handle the return value, and included an extra parameter to represent the condition code. If a particular system intrinsic does not return a value, then that parameter can be omitted. Likewise, if the condition code does not matter to you, there is no need to include a parameter for it either.

In any case the outline for the SPL interface must look like this:

```
$CONTROL SUBPROGRAM
BEGIN
PROCEDURE name your procedure and state its parameter list in
  parentheses;
ARRAY declare any arrays for future equivalencing here;
INTEGER declare any single word integers here;
DOUBLE declare any integers with 4 to 9 digits here;
BEGIN
  BYTE ARRAY name of local byte array (*)=name of passed word
    array;
  INTRINSIC names of the System Intrinsics that you want to call;
  call the System Intrinsic;
  set up the condition code;
END;
END.
```

This all looks quite complicated but be assured after the first time you´ll love it.

We have not covered all of the possibilities here. We have only tried, through a representative sample, to show what generally must take place for an interface to function properly. Perhaps one important factor that was not brought out is that any number of interface procedures can be contained in one SPL source file. Merely stick them in between the main BEGIN/END pair like this:

```
$CONTROL SUBPROGRAM
BEGIN

PROCEDURE first procedure;
BEGIN
code for first procedure;
END;

PROCEDURE next procedure;
BEGIN
code for this procedure;
END;

and so on . . .

END.
```

And there you are.

JOURNEY FROM AN ABORT MESSAGE TO A LINE OF SOURCE CODE

Madeline Lombaerde
General Systems Division

A key element in isolating program problems (user or subsystem) is the proper interpretation of an abort message. This note deals with the principal methods for going from an abort message back to the line of source code where the abort occurred. The examples for FORTRAN and BASIC compiled programs use the methods that apply to SPL as well.

While it may seem as if the same methods can be applied directly to COBOL, there are certain conventions used by COBOL that are hidden from the user and significantly complicate the search for a program bug when the methods in this note are used. Therefore, the recommended procedure for debugging COBOL for now is to put in tracing DISPLAY statements, for example, at the beginning of each paragraph and/or section. As for RPG, built-in debugging features should be used.

The following examples show two compiled programs and the steps required to identify the location of an error in the source code of each program when given an abort message.

NOTE

The figures in EXAMPLES 1 and 2 have been edited. Only data directly pertinent to the discussion of the examples is included.

Example 1 is a program (ABCINFO) compiled using FORTRAN A (figures 1A and 1B) and FORTRAN B (figures 2A and 2B).

Example 1: FORTRAN A and B

While running program ABCINFO, the following abort message occurs:

```
                  ABORT :$OLDPASS...%0.%13
                  PROGRAM ERROR #24: BOUNDS VIOLATION
```

The program PMAP is:

```
PROGRAM FILE $NEWPASS.HLBOOK.MAL

ABC1                 0
   NAME               STT  CODE ENTRY SEG
   ABCSETUP             1     0     0
   SEGMENT LENGTH            20
ABC0                 1
   NAME               STT  CODE ENTRY SEG
   ABCINFO              1     0    11
   ABCSETUP             2                0
   TFORM'               3                ?
   FMTINIT'             4                ?
   TERMINATE'           5                ?
   IIO'                 6                ?
   AIIO'                7                ?
   SEGMENT LENGTH           110

PRIMARY DB     0    INITIAL STACK    1440   CAPABILITY      600
SECONDARY DB   3    INITIAL DL          0   TOTAL CODE      130
TOTAL DB       3    MAXIMUM DATA    11610   TOTAL RECORDS     6
ELAPSED TIME 00:00:02.134            PROCESSOR TIME  00:00.446
```

The source was compiled using Fortran A with the $CONTROL options of LABEL and MAP. Using Fortran B, $CONTROL LOCATION was used. See figure 1 and 2.

```
$CONTROL USLINIT,LABEL,MAP,SEGMENT=ABC0
      PROGRAM ABCINFO
      INTEGER ABC(100)
    5 WRITE (6,600)
  600 FORMAT (" NUM? ")
      READ (5,*) NUM
      IF (NUM.EQ.0) STOP
      CALL ABCSETUP(ABC,NUM)
   10 WRITE(6,601) ABC
  601 FORMAT(10F6.2)
      GO TO 5
      END


  SYMBOL MAP

NAME             TYPE       STRUCTURE    ADDRESS

ABC              INTEGER    ARRAY        Q+  1,I
ABCSETUP                    SUBROUTINE
NUM              INTEGER    SIMPLE VAR   Q+  2


  LABEL MAP

  STATEMENT   CODE     STATEMENT    CODE     STATEMENT    CODE
    LABEL    OFFSET      LABEL     OFFSET      LABEL     OFFSET

       5        15         10          53     600 FMT        0
     601 FMT     5
```

Figure 1A.

```
00013000  $CONTROL SEGMENT=ABC1
00014000        SUBROUTINE ABCSETUP(I,N)
00015000        INTEGER I(N)
00016000        DO 1 J=1,N
00017000      1 I(J)=J
00018000      2 RETURN
00019000        END



  SYMBOL MAP

NAME                TYPE         STRUCTURE      ADDRESS

ABCSETUP                         SUBROUTINE
I                   INTEGER      ARRAY          Q-   5,I
J                   INTEGER      SIMPLE VAR     Q+   2
N                   INTEGER      SIMPLE VAR     Q-   4,I


   LABEL MAP

  STATEMENT    CODE      STATEMENT    CODE      STATEMENT    CODE
    LABEL     OFFSET       LABEL     OFFSET       LABEL     OFFSET

      1         11           2         15
```

Figure 1B.

```
PAGE 0001 HP32102B.00.09 FORTRAN/3000 (C)HEWLETT-PACKARD CO. 1976


00001000  $CONTROL USLINIT,LABEL,MAP,SEGMENT=ABC0,LOCATION
00015  00002000        PROGRAM ABCINFO
00015  00003000        INTEGER ABC(100)
00015  00004000      5 WRITE (6,600)
00032  00005000    600 FORMAT(" NUM? ")
00032  00006000        READ (5,*) NUM
00043  00007000        IF (NUM.EQ.0) STOP
00047  00008000        CALL ABCSETUP(ABC,NUM)
00053  00009000     10 WRITE(6,601) ABC
00074  00010000    601 FORMAT(10F6.2)
00074  00011000        GO TO 5
00075  00012000        END


  SYMBOL MAP

NAME          TYPE          STRUCTURE       ADDRESS

ABC           INTEGER       ARRAY           Q+  1,I
NUM           INTEGER       SIMPLE VAR      Q+  2
ABCSETUP                    SUBROUTINE


  LABEL MAP

  STATEMENT   CODE     STATEMENT   CODE     STATEMENT   CODE
    LABEL    OFFSET      LABEL    OFFSET      LABEL    OFFSET

       5       15         10        53       600 FMT      0
     601 FMT    5
```

Figure 2A.

```
PAGE 0002  HEWLETT-PACKARD 32102B.00.09  FORTRAN/3000

00004   00013000  $CONTROL SEGMENT=ABC1
00004   00014000        SUBROUTINE ABCSETUP(I,N)
00004   00015000        INTEGER I(N)
00004   00016000        DO 1 J=1,N
00011   00017000      1 I(J)=J
00015   00018000      2 RETURN
00016   00019000        END


  SYMBOL MAP

NAME               TYPE        STRUCTURE     ADDRESS

ABCSETUP                       SUBROUTINE
J                  INTEGER     SIMPLE VAR    Q+  2
I                  INTEGER     ARRAY         Q-  5,I
N                  INTEGER     SIMPLE VAR    Q-  4,I


  LABEL MAP

STATEMENT   CODE    STATEMENT   CODE
  LABEL    OFFSET     LABEL    OFFSET

     1        11        2         15
```

Figure 2B.

The place to start is the PMAP.

1. Find the name of the program unit in the segment %0 whose code begins before location %13 and whose last code location is greater than %13.

```
                          ----> SEGMENT 0
ABC1                   [0]
   NAME                 STT    CODE ENTRY SEG
  [ABCSETUP]             1       0      0 }
   SEGMENT LENGTH               20        } ----> 13 IS BETWEEN
                                                     0 AND 20
```

Now we know the abort is in the subroutine ABCSETUP.

Next,

2. Calculate the relative program unit offset (RPUO) as follows:

   RPUO=Abort location - start of code

   From this formula, our RPUO for this example is

   %13 - %0 = %13

3. Now go to the compiler listing of ABCSETUP.

   a. Fortran A (Fig. 1b): Locate the statement label whose offset is less than %13 but such that the offset for the next label in the source code is greater than %13.

      The abort occurred between line 17 (statement 1) and line 18 (statement 2).

   b. Fortran B (Fig. 2b): Locate the statement whose relative code offset is less than %13 such that the offset for the very next statement is greater than %13.

      The abort occurred while trying to execute this statement.

      A bounds violation indicates that an attempt was made to address a location outside of the code or data statement. To check why the loop was going out of bounds, the user should check the data for NUM making sure that it is less than 100. If the data is OK, DEBUG can be used to check the value of J at the time of the abort (do a :SETDUMP and then check Q+2 in ABCSETUP).

Example 2 is a program (ABCBSC) compiled using BASIC.

Example 2: BASICOMP

```
While running program ABCBSC, the following abort message occurs:


            ABORT :$OLDPASS...%0.%367:SYSL.%154.%3741
            PROGRAM ERROR #24: BOUNDS VIOLATION

  The program PMAP is

  PROGRAM FILE $NEWPASS.HLBOOK.MAL

  SEG'               0
     NAME             STT  CODE ENTRY SEG
     B'LLBL                   0     0
     B'ABORT


     TERMINATE'        12                ?
     NOECHO             3   233   236
     ABCBSC             4   242   242
     B'INPUTNUM        13                ?
     B'TERMIO          14                ?
     B'PRINTNUM        15                ?
     B'STOP            16                ?
     B'ABORTPLUSNUM    17                ?
     SEGMENT LENGTH         424

  PRIMARY DB       17    INITIAL STACK   1440   CAPABILITY     600
  SECONDARY DB    424    INITIAL DL         0   TOTAL CODE     424
  TOTAL DB        443    MAXIMUM DATA   11610   TOTAL RECORDS   11
  ELAPSED TIME     00:00:03.708            PROCESSOR TIME  00:00.621
```

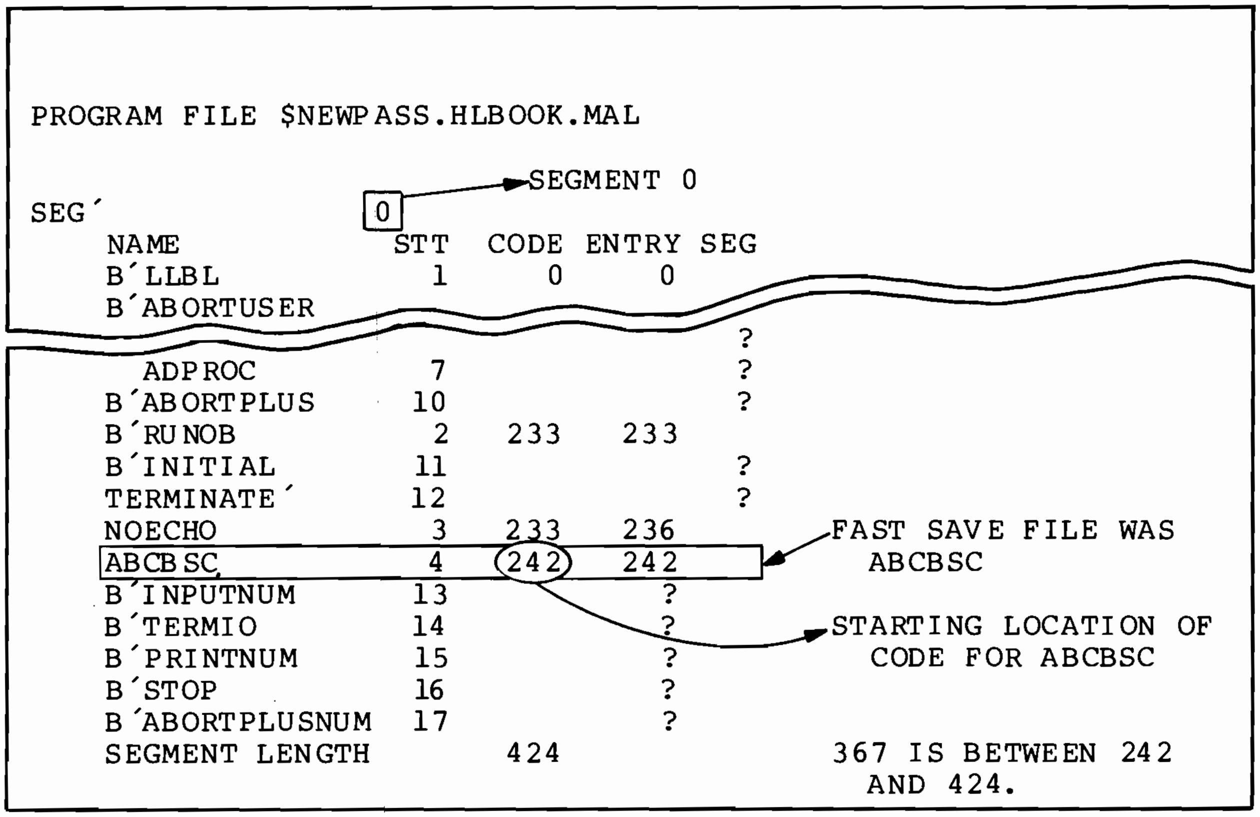
The Basic Fast Save file was compiled with $CONTROL SOURCE,LABEL, MAP

```
  PAGE 0001  HEWLETT-PACKARD 32103B.00.08(4WD)  BASICOMP


  $CONTROL USLINIT,SOURCE,MAP,LABEL
  $COMPILE ABCBSC
    10 DIM A[50]
    20 INPUT "I=",I
    30 IF I=0 THEN STOP
    40 FOR J=1 TO I
    50   A[J]=J
    60 NEXT J
    70 PRINT (FOR J=1 TO I,A[J])
    80 GOTO 20


      SYMBOL MAP

      ABCBSC

  NAME        TYPE          STRUCTURE          ADDRESS

   A          REAL          ARRAY              Q+  4,I
   I          REAL          SIMPLE VAR         Q+  5
   J          REAL          SIMPLE VAR         Q+  7


LABEL MAP     ENTRY POINT      0  STACK  111
 LABEL LOCATION  LABEL LOCATION  LABEL LOCATION   LABEL LOCATION
   10= 20     41    20         41    30          54    40         61
   50         73    60        100    70         106    80= 20    136
```

Figure 3

Again, start with the PMAP:

1. Locate the program unit in Segment zero whose code begins before location %367 and whose last code location is greater than %367. This is in segment %0. The reason we go to %0.%367 instead of to location %154.%3741 is because this library (SL.PUB.SYS) routine is called at location %0.%367 and probably received bad data to begin with.



```
PROGRAM FILE $NEWPASS.HLBOOK.MAL

                                  SEGMENT 0
SEG´                   0
    NAME                STT   CODE  ENTRY  SEG
    B´LLBL                1     0      0
    B´ABORTUSER
                                             ?
       ADPROC             7                  ?
    B´ABORTPLUS          10                  ?
    B´RUNOB               2    233    233
    B´INITIAL            11                  ?
    TERMINATE´           12                  ?
    NOECHO                3    233    236        FAST SAVE FILE WAS
    ABCBSC                4    242    242          ABCBSC
    B´INPUTNUM           13             ?
    B´TERMIO             14             ?        STARTING LOCATION OF
    B´PRINTNUM           15             ?          CODE FOR ABCBSC
    B´STOP               16             ?
    B´ABORTPLUSNUM       17             ?
    SEGMENT LENGTH             424               367 IS BETWEEN 242
                                                   AND 424.
```

2. Now we know that the abort occurred in ABCBSC and we can calculate the RPUO as before:

   RPUO=Abort location - start of code

   RPUO=%367 - %242 = %125

3. Now go to the compiler listing (Fig. 3): look for the statement whose starting location is less than %125 but such that the next sequential statement has a beginning location greater than %125.

   The abort occured during execution of statement 70. The data for I should be checked to assure that the value of I is less than or equal to 50.

RIN'S

Madeline Lombaerde/Hal Goodwin
General Systems Division

Several requests have been received asking for an example of a way to use global RIN´s. In providing the following example, we assume that the reader has some familiarity with the concept of RIN´s. Section VI of the MPE Intrinsics reference manual should be reviewed as background for this article.

LOCKING AND UNLOCKING GLOBAL RIN´S

Any global RIN assigned to a group of users can be locked by one job at a time with the LOCKGLORIN intrinsic. Once a RIN is locked, any other jobs that attempt to lock this RIN are suspended.

In order to lock a global RIN, you must know: 1) the RIN number returned by MPE when the RIN was acquired with the :GETRIN command, and 2) the password which was specified in the rinpassword parameter of the :GETRIN command. If you are a user with only the standard MPE capability, you can lock only one global RIN at a time.

USING RIN´S TO MANAGE FILE RECORDS

Most users are aware that a file can be locked in order to prevent concurrent access to a file by other processes trying to lock the same file. This allows a user to make sure, for instance, that a part of the file is not read while an update is being performed. This file locking is accomplished through the mutual cooperation of all processes accessing the file by means of the FLOCK and FUNLOCK intrinsics.

There may be times, however, when a file is large enough and enough processes need to access it regularly, that locking the entire file can result in a significant amount of lost time. This happens when all the other processes must remain suspended until the file is available.

One possible solution is to lock only part of the file, allowing processes which plan to access a different part of the file to continue processing. Only those processes attempting to access the "locked" records would be suspended.

Figure A contains a program which uses the LOCKGLORIN and UNLOCKGLORIN intrinsics. The program allows a user to lock four records, as a RIN, in a file so that a record can be updated without any chance of another user updating the same record simultaneously. Additionally, the other users are not suspended when attempting to access and update records elsewhere in the file.

The file used in the example below contains 20 records and therefore five contiguous RIN´s have to be acquired (there are four records per RIN) before the program is run. This is accomplished by entering five :GETRIN commands as follows:

```
  :GETRIN BOOKRIN
```

where BOOKRIN is specified as the rinpassword parameter. BOOKRIN is the password which is used in the program to lock the RIN (see statements 6 and 36 in figure A).

```
TITLE:   THE BORROWERS              LOCN:   AVAILABLE
TITLE:   ALICE IN WONDERLAND        LOCN:   AVAILABLE
TITLE:   PETER PAN                  LOCN:   AVAILABLE
TITLE:   JUNGLE BOOK                LOCN:   AVAILABLE
TITLE:   MARY POPPINS               LOCN:   AVAILABLE
TITLE:   TOM SAWYER                 LOCN:   AVAILABLE
TITLE:   TREASURE ISLAND            LOCN:   AVAILABLE
TITLE:   A CHRISTMAS CAROL          LOCN:   AVAILABLE
TITLE:   HOUSE AT POOH CORNER       LOCN:   AVAILABLE
TITLE:   THE WIZARD OF OZ           LOCN:   AVAILABLE
TITLE:   SLEEPING BEAUTY            LOCN:   AVAILABLE
TITLE:   TALES OF MOTHER
         GOOSE                      LOCN:   AVAILABLE
TITLE:   AESOP´S FABLES             LOCN:   AVAILABLE
TITLE:   KIDNAPPED                  LOCN:   AVAILABLE
TITLE:   OLIVER TWIST               LOCN:   AVAILABLE
TITLE:   DR. DOOLITTLE              LOCN:   AVAILABLE
TITLE:   WHEN WE WERE
         VERY YOUNG                 LOCN:   AVAILABLE
TITLE:   H.M.S. PINAFORE            LOCN:   AVAILABLE
TITLE:   WORLD BOOK
         ENCYCLOPEDIA               LOCN:   AVAILABLE
TITLE:   COLLEGIATE
         DICTIONARY                 LOCN:   AVAILABLE
```

The program in figure A establishes the RIN number limits 2 and 6 (see statement number 14), thus using only RIN numbers 2, 3, 4, 5, and 6. MPE returns the RIN number assigned each time the :GETRIN command is entered. Because MPE does not always assign RIN numbers in sequence, however, it may be necessary to enter more than five :GETRIN commands in order to acquire the five contiguous RIN´s 2, 3, 4, 5, and 6. Extra RIN´s can be released with the :FREERIN command.

The statements

```
  FWRITE(OUT,REQUEST,8,%320);CCNE(5);
```

request a book number from the user and perform a condition code check. Note that in statement number 16, CCNE has been defined as

```
--
  IF<>THEN QUIT#
```

```
$CONTROL USLINIT,INNERLIST
BEGIN
  BYTE ARRAY INPUT(0:5):="INPUT ";
  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
  BYTE ARRAY NAME(0:8):="BOOKFILE ";
  BYTE ARRAY PASSWD(0:7):="BOOKRIN ";
  INTEGER IN,OUT,BOOK,LGTH,ACCNO,RIN;
  LOGICAL DUMMY,COND:=TRUE;
  ARRAY BUFR(0:35);
  BYTE ARRAY BBUFR(*)=BUFR;
  ARRAY HEAD(0:13):="LIBRARY INFORMATION PROGRAM.";
  ARRAY REQUEST(0:7):=%6412,"ACCESSION NO: ";
  ARRAY CHANGE(0:9):="      NEW LOCATION: ";
  EQUATE RINBASE=2, RECDS'PER'RIN=4, MAXRIN=6;
  DEFINE CCL =IF < THEN QUIT#,
         CCNE=IF <> THEN QUIT#;

  INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,FREADDIR,FWRITEDIR,
            LOCKGLORIN,UNLOCKGLORIN,QUIT,BINARY;

  <<END OF DECLARATIONS>>

      IN:=FOPEN(INPUT,%45); CCL(1);                     <<$STDIN>>
      OUT:=FOPEN(OUTPUT,%414); CCL(2);                  <<$STDLIST>>
      BOOK:=FOPEN(NAME,%5,%304); CCL(3);                <<OLD DISC FILE>>
      FWRITE(OUT,HEAD,14,0); CCNE(4);                   <<PROGRAM ID>>
LOOP:
      FWRITE(OUT,REQUEST,8,%320); CCNE(5);              <<REQST BOOK NUMBR>>
      LGTH:=FREAD(IN,BUFR,-10); CCNE(6);                <<INPUT NUMBER>>
      IF LGTH=0 THEN GO EXIT;                           <<NO INPUT-EXIT>>
      ACCNO:=BINARY(BBUFR,LGTH);                        <<CONVERT NUMBER>>
      IF <> THEN GO LOOP;                               <<IF BAD TRY AGAIN>>

      RIN:=RINBASE+(ACCNO/RECDS'PER'RIN);               <<COMPUTE RIN NO.>>
      IF NOT(RINBASE<=RIN<=MAXRIN) THEN GO LOOP;        <<BOUNDS CHECK RIN>>
      LOCKGLORIN(RIN,COND,PASSWD);                      <<LOCK FILE SUBSET>>

      FREADDIR(BOOK,BUFR,36,DOUBLE(ACCNO)); CCL(7);     <<READ BOOK DATA>>
      IF > THEN GO AGAIN;                               <<EOF - TRY AGAIN>>
      FWRITE(OUT,BUFR,36,0); CCNE(8);                   <<DISPLAY DATA>>
      FWRITE(OUT,CHANGE,10,%320); CCNE(9);              <<REQST A CHANGE>>

      BUFR(19):="  ";
      MOVE BUFR(20):=BUFR(19),(16);                     <<BLANK OLD LOCN>>
LGTH:=FREAD(IN,BUFR(19),17); CCNE(10);                  <<READ NEW LOCN>>
      IF LGTH>0 THEN                                    <<NEW LOCN ENTERED>>
        BEGIN
          FWRITEDIR(BOOK,BUFR,36,DOUBLE(ACCNO));        <<MODIFY THE FILE>>
          CCNE(11);                                     <<CHECK FOR ERROR>>
        END;
      FCONTROL(BOOK,2,DUMMY); CCL(12);                  <<FORCE RECD POST>>
AGAIN:
      UNLOCKGLORIN(RIN); CCNE(13);                      <<UNLOCK SUBSET>>
      GO LOOP;                                          <<CONTINUE>>
EXIT:END.
```

Figure A. Using the LOCKGLORIN and UNLOCKGLORIN Intrinsics

This eliminates the need to repeat the entire statement at every point in the program where such a condition code check is required. Instead, the statement CCNE and an arbitrary number (5 in this case) can be used.

The book number is read with the statement

```
  LGTH:=FREAD(IN,BUFR,-10);
```

and converted to a binary value with the statement

```
  ACCNO:=BINARY(BBUFR,LGTH);
```

The RIN number to be locked is computed with the statement

```
  RIN:=RINBASE+(ACCNO/RECDS'PER'RIN);
```

RINBASE and RECDS'PER'RIN have been equated to 2 and 4, respectively (see statement number 14). Thus, if book number 3 is entered by the user, the RIN number to be locked would be computed as RIN number 2, as follows:

```
  RIN = 2+(3/4)
      = 2+0(integer division)
```

The record specified by the book number is displayed for the user and the change ("NEW LOCATION: ") is requested. The existing location information is filled with blanks with the statements

```
  BUFR(19):=" ";
  MOVE BUFR(20):=BUFR(19),(16);
```

The new location is entered and read with the statement

```
  LGTH:=FREAD(IN,BUFR(19),17);
```

and the record is updated with the statement

```
  FWRITEDIR(BOOK,BUFR,36,DOUBLE(ACCNO));
```

The statement

```
  FCONTROL(BOOK,2,DUMMY);
```

is used in case the file which has been opened is a buffered file. This statement insures that the process' buffers are posted to the disc before the RIN is unlocked.

Note that in a program of this kind, it is important that the number of records per block and the number of records per RIN are the same. The RIN must contain a complete block of records.

The statement

```
  UNLOCKGLORIN(RIN);
```

unlocks the RIN before the loop is repeated. When the user enters a new book number, a new RIN number will be computed and that RIN number will be locked.

When a carriage return is entered, signifying no input, the program terminates.

The results of running the program and the updated condition of the library file are shown below.

```
:RUN LIBIN


LIBRARY INFORMATION PROGRAM


ACCESSION NO: 3
TITLE: JUNGLE BOOK    LOCN: AVAILABLE
    NEW LOCATION: FACULTY LOAN - DR.
     SCHWARTZ

ACCESSION NO: 10
TITLE: SLEEPING BEAUTY LOCN: AVAILABLE
    NEW LOCATION: LOANED CARD#451, DUE
     JUNE 6

ACCESSION NO: 3
TITLE: JUNGLE BOOK     LOCN: FACULTY LOAN -
                       DR: SCHWARTZ
    NEW LOCATION:

ACCESSION NO: 9
TITLE: THE WIZARD OF   LOCN: AVAILABLE
 OZ
    NEW LOCATION: INTERLIBRARY LOAN -
     UNIV. of OZ

ACCESSION NO: 3
TITLE: JUNGLE BOOK     LOCN: FACULTY LOAN -
                       DR. SCHWARTZ
    NEW LOCATION: AVAILABLE

ACCESSION NO:

END OF PROGRAM


TITLE:  THE BORROWERS          LOCN:  AVAILABLE
TITLE:  ALICE IN WONDERLAND    LOCN:  AVAILABLE
TITLE:  PETER PAN              LOCN:  AVAILABLE
TITLE:  JUNGLE BOOK            LOCN:  AVAILABLE
TITLE:  MARY POPPINS           LOCN:  AVAILABLE
TITLE:  TOM SAWYER             LOCN:  AVAILABLE
```

TITLE:	TREASURE ISLAND	LOCN:	AVAILABLE
TITLE:	A CHRISTMAS CAROL	LOCN:	AVAILABLE
TITLE:	HOUSE AT POOH CORNER	LOCN:	AVAILABLE
TITLE:	THE WIZARD OF OZ	LOCN:	INTER-LIBRARY LOAN - UNIV. OF OZ
TITLE:	SLEEPING BEAUTY	LOCN:	LOANED CARD #451, DUE JUNE 6
TITLE:	TALES OF MOTHER GOOSE	LOCN:	AVAILABLE
TITLE:	AESOP´S FABLES	LOCN:	AVAILABLE
TITLE:	KIDNAPPED	LOCN:	AVAILABLE
TITLE:	OLIVER TWIST	LOCN:	AVAILABLE
TITLE:	DR. DOOLITTLE	LOCN:	AVAILABLE
TITLE:	WHEN WE WERE VERY YOUNG	LOCN:	AVAILABLE
TITLE:	H.M.S. PINAFORE	LOCN:	AVAILABLE
TITLE:	WORLD BOOK ENCYCLOPEDIA	LOCN:	AVAILABLE
TITLE:	COLLEGIATE DICTIONARY	LOCN:	AVAILABLE

CALLING SPL FROM RPG AND CALLING COBOL FROM SPL

Bruce Campbell
HP Neely Sales Region

I. When you call SPL from RPG using RLABL and EXIT, the RPG compiler generates the equivalent of a global byte pointer declaration for fields used in an RLABL operation. For example:

```
          RLABL XYZ
```

generates the equivalent of:

```
          global byte pointer XYZ;
```

To access parameters declared with the RLABL operation from an SPL procedure, the parameter should be declared in the local declaration section of the procedure as "EXTERNAL BYTE POINTER". For example, given the following RPG code:

```
          RLABL XYZ
          EXIT EXTPRO
```

The SPL procedure would start:

```
          PROCEDURE EXTPRO;
          BEGIN
              EXTERNAL BYTE POINTER XYZ;
```

Note the following points:

1. It does not matter whether the RPG field named in an RLABL operation is alpha or numeric (i.e., packed). Both are referenced as external byte pointers from a called SPL procedure.

2. Parameters passed to SPL procedures by RLABL are not defined in the procedure head, but only in the local declarations.

Below is an RPG program, RPGDATE, that calls an SPL procedure, MDYCON, and passes it two files, MDYRZZ and JDATZZ, through the RLABL operation.

II. In calling COBOL from SPL there are three items to keep in mind:

1. Compile the COBOL program with the DYNAMIC option in the $CONTROL statement. This forces the COBOL compiler to allocate all data division storage Q-RELATIVE.

2. Specify the parameters passed to the COBOL program in both the linkage section and in the USING option of the procedure statement. For example:

```
LINKAGE SECTION.
01 PARAM1 PIC S9(5) COMP-3.
01 PARAM2 PIC S9(4) COMP.
PROCEDURE DIVISION USING PARAM1 PARAM2
```

The parameters must be specified in the USING option of the Procedure Division statement in the same order that they are given in the SPL call to the program. Also, the parameters must be declared in the linkage section at the 01 level.

3. All parameters passed to a COBOL program must be word-addressed since the COBOL compiler always generates word addresses for external data items (as distinguished from byte addresses). Also, the COBOL compiler always starts 01-LEVEL COMP-3 fields in the left-hand byte of a word. This means that the declaration

```
01 FIELD PIC S9(5) COMP-3.
```

which uses two words, has a slack byte in the right-most byte of those two words. COBOL always handles parameters by reference, but file number is passed from COBOL by value.

III. Below are three example programs.

1. RPGDATE: An RPG program that calls an SPL routine called MDYCON and makes two packed decimal data fields available to it.

2. MDYCON: An SPL program that calls a COBOL routine called MDYCONC. MDYCON takes care of the addressing incompatibility between the RPG byte pointers and the COBOL word addresses.

3. MDYCONC: Translates the MM/DD/YY date passed from the RPG program to Julian format and returns the Julian date to the RPG program.

RPGDATE

```
$TITLE"***TEST DATE CONVERSION:
MM/DD/YY TO JULIAN*****"
H
FINFILE   IPEAF 72 72  DISC
FOUTFILE  O    F 72 72  DISC
IINFILE   AA 01
I                           1 60DATEIN
C      RLABL            JDATZZ   50
```

```
C       RLABL           MDYRZZ  60
C       Z-ADDDATEIN     MDYRZZ
C       EXIT MDYCON
OOUTFILE  D   01
O                       JDATZZX  10
```

MDYCON

```
$CONTROL SUBPROGRAM
 BEGIN
 PROCEDURE MDYCON:
  BEGIN
  <<
  ***ALIGN PARAMETERS ON WORD BOUNDARIES
  ***AND PASS WORD ADDRESS TO COBOL
  >>
  EXTERNAL BYTE POINTER MDYRZZ,JDATZZ;
  LOGICAL ARRAY MDYR(0:1),JUL(0:1);
  BYTE ARRAY BMDYR(*)=MDYR,BJUL(*)=JUL;
  PROCEDURE MDYCONC(DATE´IN,DATE´OUT);
   INTEGER ARRAY DATE´IN,DATE´OUT;
   OPTION EXTERNAL;
  MOVE BMDYR:=MDYRZZ,(4);
  MDYCONC(MDYR,JUL);
  MOVE JDATZZ:=BJUL,(3);
  END;
 END;
```

MDYCONC1

```
$TITLE "*********CONVERT STANDARD TO JULIAN************"
$CONTROL DYNAMIC,MAP
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MDYCONC.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.
01  UNPAK-DATE PIC9(6).
01  DATE-SUBS.
    02 MM PIC 99.
    02 DD PIC 99.
    02 YY PIC 99.
01  MONTH-DISPLACEMENT-TABLE.
    02 MONTH-DISP PIC 999 OCCURS 12 TIMES.
01  QUOT PIC S999.
01 REM PIC S9.
```

```
LINKAGE SECTION.
01 MDY-DATE PIC S9(6) COMP-3.
01  JULIAN-DATE PIC S9(5) COMP-3.
*****

PROCEDURE DIVISION USING MDY-DATE JULIAN-DATE.
START-MDYCON SECTION.
   DISPLAY "RECEIVED " MDY-DATE.
   MOVE "000031059090120151181212243273304334"
    TO MONTH-DISPLACEMENT-TABLE.
   MOVE MDY-DATE TO UNPAK-DATE.
   MOVE UNPAK-DATE TO DATE-SUBS.
   DISPLAY "Y,M,D:" YY ";" MM ";" DD ";".
   IF YY=0 OR MM=0 OR MM>12 OR DD=0 OR DD>31 THEN
    MOVE 99999 TO JULIAN-DATE
    DISPLAY "FLUSHED WITH " JULIAN-DATE
    GOBACK.
   COMPUTE JULIAN-DATE = YY*1000.
   COMPUTE JULIAN-DATE = JULIAN-DATE + MONTH-DISP(MM).
   COMPUTE JULIAN-DATE = JULIAN-DATE + DD.
   DISPLAY "SENT BACK " JULIAN-DATE.
   DIVIDE 4 INTO YY GIVING QUOT REMAINDER REM.
   IF REM NOT = ZERO THEN GOBACK.
   IF MM>2 COMPUTE JULIAN-DATE = JULIAN-DATE + 1.
   GOBACK.
```

MDYCONC2
Symbol Table Map

```
LVL SOURCE NAME                  BASE   DISPL   SIZE     USAGE    CATEGORY R O D J

WORKING-STORAGE SECTION

01 UNPAK-DATE                    Q      000174  000006   DISP     DISP-N
01 DATE-SUBS                     Q      00C202  000006            GROUP
02 MM                            Q      000202  000002   DISP     DISP-N
02 DD                            Q      000204  000002   DISP     DISP-N
02 YY                            Q      000206  000002   DISP     DISP-N
01 MONTH-DISPLACEMENT-TABLE      Q      000210  000044            GROUP
02 MONTH-DISP                    Q      000210  000044   DISP     DISP-N     O
01 QUOT                          Q      000254  000003   DISP     DISP-NS
01 REM                           Q      000260  000001   DISP     DISP-NS

LINKAGE SECTION

01 MDY-DATE                      LINK   000000  000004   COMP-3   N
01 JULIAN-DATE                   LINK   000000  000003   COMP-3   N
```

PMAP

```
RPG´01                0
   NAME                STT   CODE   ENTRY   SEG
   RPG´08                1      0       0
   R´OUT                 2                     1
   R´INT                 3                     1
   R´CALT                4                     1
   R´CALD                5                     1
   R´CNTL                6                     ?
   SEGMENT LENGTH              34
RPG´00                1
   NAME                STT   CODE   ENTRY   SEG
   R´OUT                 1      0       0
   R´WRITE               1                     ?
   R´CALT                2     33      33
   R´CALD                3     37      37
   MDYCON                7                     2
   R´INT                 4     65      65
   R´ERROR              10                     ?
   I´0001                5     65     110
   SEGMENT LENGTH             150
RL SEGMENT            2
   MDYCON                1      0       0
   MDYCONC               2     31      31
   C´LIT´FIG´MOVE        4                     ?
   C´DECABS              5                     ?
   MPYD                  6                     ?
   C´DISPLAY´INIT        7                     ?
   C´DISPLAY´L          10                     ?
   C´DISPLAY´ID         11                     ?
   C´DISPLAY´FIN        12                     ?
   DIVD                 13                     ?
   MDYCONC´              3   1154    1154
   SEGMENT LENGTH            1420
```

NOTE: This order form is for *updates only*. To order complete manuals (new, new editions, reprints), use the Corporate Parts Center order form. After being incorporated into a manual through reprinting, updates continue to be available for six months.

HEWLETT PACKARD

SOFTWARE/PUBLICATIONS DISTRIBUTION
ORDER FORM

UPDATES TO 3000 AND 2000
LEVEL MANUALS ONLY

SHIP TO:

NAME ______________________________

COMPANY ______________________________

STREET ______________________________

CITY ____________ STATE ________ ZIP CODE ________

MANUAL NAME	PART NUMBER	QUANTITY

When completed, please mail this form to:

There is no charge for manual updates.

HEWLETT-PACKARD
SOFTWARE/PUBLICATIONS DISTRIBUTION
5303 Stevens Creek Blvd.
Santa Clara, CA 95050

Please photocopy this order form if you do not want to cut the page off. You will automatically receive a new order form with your order.

HEWLETT hp PACKARD

CONTRIBUTED SOFTWARE
Direct Mail Order Form

NOTE: No direct mail order can be shipped outside the United States.

Please Print:

Name ______________________ **Title** ______________

Company ______________________________________

Street ____

City ______________ **State** __________ **Zip Code** ________

Country ______________________________________

Item No.	Part No.	Qty.	Description	List Price Each		Extended Total	

Sub-total		
Your State & Local Sales Taxes*		
Handling Charge	1	50
TOTAL		

*Tax is verified by computer according to your ZIP CODE. If no sales tax is added, your state exemption number must be provided: # ___________.
If not, your order may have to be returned.

Domestic Customers: Cash required on all orders less than $50.00. Mail the order form with your check or money order (payable to Hewlett-Packard Co.) or your U.S. Company Purchase Order to:

HEWLETT-PACKARD COMPANY
Contributed Software
P.O. Box 61809
Sunnyvale, CA 94088

International Customers: Order through your local Hewlett-Packard Sales office. No direct mail order can be shipped outside the United States.

All prices domestic U.S.A. only. Prices are subject to change without notice.

HEWLETT-PACKARD
COMPUTER SYSTEMS COMMUNICATOR ORDER FORM

Please Print:

Name ______________________ **Date** ______________________

Company ______________________

Street ______________________

City ______________ **State** ______________ **Zip Code** ______________

Country ______________________

☐ HP Employee   Account Number ______________   Location Code ______________

☐ **DIRECT SUBSCRIPTION**

Part No.	Description	Qty	List Price	Extended Dollars	Total Dollars
5951-6111	COMMUNICATOR 1000	______	$48.00	______	
	(if quantity is greater than 1 discount is 40%)			______	
	TOTAL DOLLARS for 5951-6111				______
5951-6112	COMMUNICATOR 2000	______	25.00	______	
	(if quantity is greater than 1 discount is 40%)			______	
	TOTAL DOLLARS for 5951-6112				______
5951-6113	COMMUNICATOR 3000	______	48.00	______	
	(if quantity is greater than 1 discount is 40%)			______	
	TOTAL DOLLARS for 5951-6113				______

☐ **BACK ISSUE ORDER FORM (cash only in U.S. dollars)**
(subject to availability)

Part No.	Description	Issue No.	Qty	List Price	Extended Dollars	Total Dollars
5951-6111	COMMUNICATOR 1000	______	______	$10.00	______	
		______	______	10.00	______	
		______	______	10.00	______	
	TOTAL DOLLARS					______
5951-6112	COMMUNICATOR 2000	______	______	$ 5.00	______	
		______	______	5.00	______	
		______	______	5.00	______	
	TOTAL DOLLARS					______
5951-6113	COMMUNICATOR 3000	______	______	$10.00	______	
		______	______	10.00	______	
		______	______	10.00	______	
	TOTAL DOLLARS					______
TOTAL ORDER DOLLAR AMOUNT						______

☐ **SERVICE CONTRACT CUSTOMERS**

You will receive one copy of either COMMUNICATOR 1000, 2000, or 3000 as part of your contract. Indicate additional copies below and have your local office forward. Billing will be included in normal contract invoices.

Number of additional copies __________

FOR HP USE ONLY

CONTRACT KEY

5951-6111	Number of additional copies	______
5951-6112	Number of additional copies	______
5951-6113	Number of additional copies	______

Approved ______________________

HEWLETT-PACKARD
COMMUNICATOR SUBSCRIPTION AND ORDER INFORMATION

The Computer Systems COMMUNICATORS are systems support publications available from Hewlett-Packard on an annual subscription.

The following instructions are for customers who do not have Software Service Contracts.

1. Complete name and address portion of order form.
2. For new direct subscriptions (see sample below):
   a. Indicate which COMMUNICATOR publication(s) you wish to receive.
   b. Enter number of copies per issue under Qty column.
   c. Extend dollars (quantity x list price) in Extended Dollars column.
   d. Enter discount dollars on line under Extended Dollars. (If quantity is greater than 1 you are entitled to a 40% discount.*)
   e. Enter Total Dollars (subtract discount dollars from Extended List Price dollars).

   **To qualify for discount all copies of publications must be mailed to same name and address and ordered at the same time.*

SAMPLE

[X] **DIRECT SUBSCRIPTION**

Part No.	**Description**	**Qty**	**List Price**	**Extended Dollars**	**Total Dollars**
5951-6111	COMMUNICATOR 1000	3	$48.00	$144.00	
	(if quantity is greater than 1 discount is 40%)			57.60	
	TOTAL DOLLARS for 5951-6111				$86.40

3. To order back issues (see sample below):
   a. Indicate which publication you are ordering.
   b. Indicate which issue number you want (check availability in latest COMMUNICATOR).
   c. Enter number of copies per issue.
   d. Extend dollars for each issue.
   e. Enter total dollars for back issues ordered.

   All orders for back issues of the COMMUNICATORS are cash only orders (U.S. dollars only) and are subject to availability.

SAMPLE

[X] **BACK ISSUE ORDER FORM** (cash only in U.S. dollars)
(subject to availability)

Part No.	**Description**	**Issue No.**	**Qty**	**List Price**	**Extended Dollars**	**Total Dollars**
5951-6111	COMMUNICATOR 1000	XX	1	$10.00	$10.00	
		XX	2	10.00	20.00	
				10.00		
	TOTAL DOLLARS					$30.00

4. Domestic Customers: Mail the order form with your U.S. Company Purchase Order or check (payable to Hewlett-Packard Co.) to:

   HEWLETT-PACKARD COMPANY
   Computer Systems COMMUNICATOR
   P.O. Box 61809
   Sunnyvale, CA 94088
   U.S.A.

5. International Customers: Order by part number through your local Hewlett-Packard Sales Office.

USE THIS FORM TO ORDER MANUALS

Do not order updates separately. Existing updates are automatically included in shipments. Only the current edition of a manual may be ordered.

HEWLETT hp PACKARD

CORPORATE PARTS CENTER

Direct Mail
Parts and Supplies Order Form

SHIP TO:

NAME ______________________________

COMPANY ______________________________ CUSTOMER REFERENCE # ____________

STREET ______________________________ TAXABLE*? ____________

CITY ______________ STATE ____________ ZIP CODE ____________

Item No.	Check Digit	Part No.	Qty.	Description	List Price Each		Extended Total	

Special Instructions

Sub-total		
Your State & Local Sales Taxes*		
Handling Charge	1	50
TOTAL		

*Tax is verified by computer according to your ZIP CODE. If no sales tax is added, your state exemption number must be provided: #____________
If not, your order may have to be returned.

Check or Money Order, made payable to Hewlett-Packard Company, must accompany order.

When completed, please mail this form with payment to:

HEWLETT-PACKARD COMPANY
Mail Order Department
P.O. Drawer #20
Mountain View, CA 94043

Phone: (415) 968-9200

Most orders are shipped within 24 hours of receipt. Shipments to California, Oregon and Washington will be made via UPS. Other shipments will be sent Air Parcel Post, with the exception that shipments over 25 pounds will be made via truck. No Direct Mail Order can be shipped outside the U.S.

Printed in U.S.A. 10/78
Part No. 5951-6113