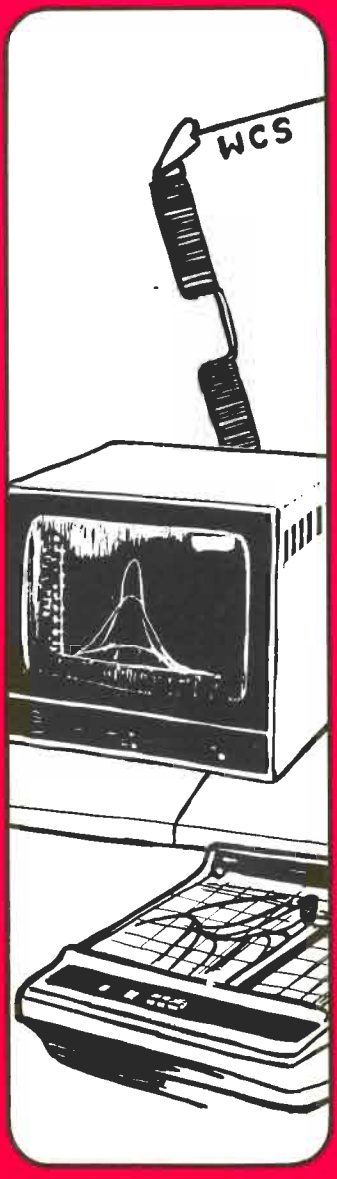
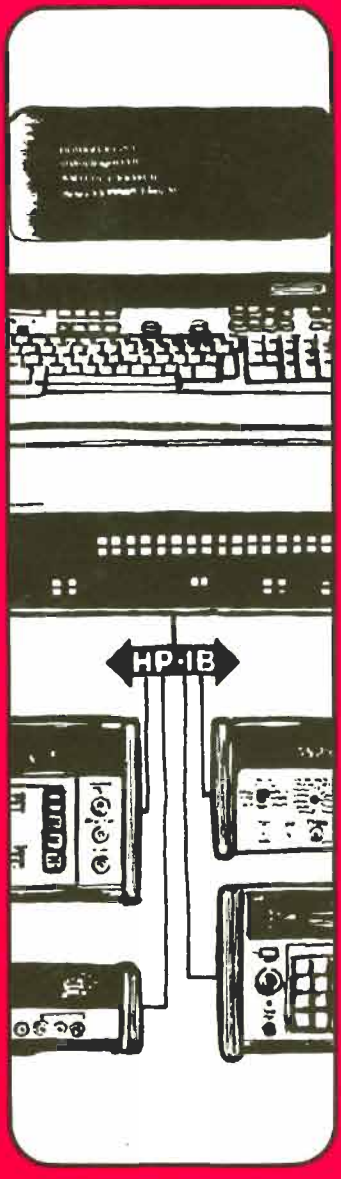
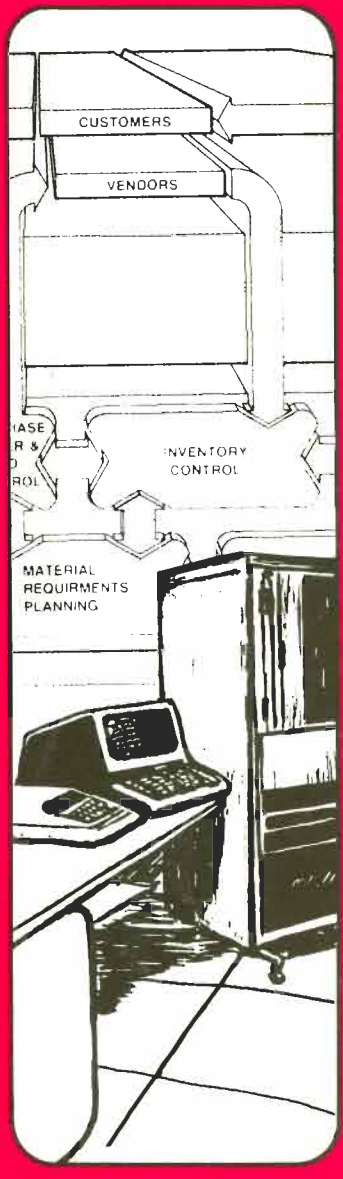
HEWLETT PACKARD

Hewlett-Packard Computer Systems

COMMUNICATOR







1980

Volume IV Issue 1

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HEWLETT-PACKARD
COMPUTER SYSTEMS

Volume IV
Issue 1

COMMUNICATOR/1000

Feature Articles

OPERATIONS MANAGEMENT — 18 **USE OF THE 264X TERMINAL AS A DATA RECORDER**
C. L. Elliot and J. L. McLaughlin/Elliot Geophysical Co.

LANGUAGES — 24 **AN OVERVIEW OF THE PASCAL LANGUAGE**
Van Diehl/HP Data Systems Division

OPERATING SYSTEMS — 34 **HP SUBROUTINE LINKAGE CONVENTIONS**
Robert Niland/HP Lexington

OEM CORNER — 42 **A BRIEF INTRODUCTION TO THE C PROGRAMMING LANGUAGE**
Tim Chase/Corporate Computer Systems, Inc.

Departments

EDITOR'S DESK — 2 **ABOUT THIS ISSUE**
3 **BECOME A PUBLISHED AUTHOR IN THE COMMUNICATOR/1000**

USER'S QUEUE — 5 **LETTERS TO THE EDITOR**

BIT BUCKET — 7 **WHO OWNS THAT TYPE 6 FILE?**
12 **CRT SCREEN ENHANCEMENTS**
13 **HOW TO PREVENT DS/1000 MONITORS FROM HOGGING PARTITIONS**

BULLETINS — 52 **A NEW HP-IB COURSE**
53 **A NEW TOOL FOR THE UNDERSTANDING OF RTE INTERNALS — RTE LISTINGS**
55 **PRESENT A PAPER AT THE FIRST HP/1000 INTERNATIONAL USERS GROUP CONFERENCE**
57 **PROGRAM OPTIMIZATION**
62 **JOIN AN HP/1000 USER GROUP!**
64 **THE PLUS/1000 LIBRARY NOW HAS A HOME!**

EDITOR'S DESK

ABOUT THIS ISSUE

This, the first Communicator issue of 1980, introduces a slightly different format. It is the hope of everyone involved in putting together this journal, that you will like the new format, and will continue to give us the same outstanding support that we enjoyed in 1979.

This issue of the Communicator could be called a Languages issue. Two of the four feature articles in this issue deal with languages. The first is titled "An Overview of the Pascal Language", and is contributed by Van Diehl, of HP's Data Systems Division. This article is the first of a two part series. The first part is intended to provide a general introduction to the Pascal language and structured programming. The second will deal with details about HP's new PASCAL compiler. The second language oriented article is titled, "A Brief Introduction to the C Programming Language". You will notice that this article is not included in the Languages feature article category. Hewlett Packard and Corporate Computer Systems jointly developed a C compiler, and this article was provided to the Communicator by Tim Chase of CCS. This article is the first contribution in a long while intended for the OEM corner. Hopefully this contribution is the first of several OEM contributions that will occur during 1980.

In the Operations Management category, C. L. Elliot, of Elliot Geophysical, in Tuscon, Arizona, has contributed an article which he calls, "Use of the 264X Terminal as a Data Recorder". I am certain that Communicator readers will find Mr. Elliot's ideas not only clever, but also useful in many varied applications.

The fourth feature article in this issue is Part II of the series started in the last issue, titled "HP Subroutine Linkage Conventions". You will recall that this series was put together by Bob Niland, of HP's Lexington, Kentucky office. Part two of the series deals with elementary subroutine structures. Never before have HP/1000 users seen this sort of information collected together all in one place. Bob's contibution, with all of its sections combined, will make useful addition to everyone's library of manuals.

As all Communicator readers are hopefully aware, HP-32E calculators are awarded to authors of the best contributions in three categories; customers, HP field personnel, and HP factory personnel. The calculator winner for the first issue of 1980 is listed below. (Unfortunately, none of the other three authors were eligible to win.) Congratulations!

The best feature article contributed by a customer	**USE OF THE 264X TERMINAL AS A DATA RECORDER** C. L. Elliot

Again, thanks to everyone for their support in 1979, we all hope that you make 1980 the best year for the Communicator/1000 yet!

Sincerely,

The Editor

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

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

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

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

WIN AN HP-32E CALCULATOR!

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

1. HP 1000 Customers;

2. HP field employees;

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

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

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

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

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

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

   OPERATING SYSTEMS
   DATA COMMUNICATIONS
   INSTRUMENTATION
   COMPUTATION
   OPERATIONS MANAGEMENT

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

EDITOR'S DESK

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

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

A SPECIAL DEAL IN THE OEM CORNER

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

IF YOU'RE PRESSED FOR TIME . . .

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

THE MECHANICS OF SUBMITTING AN ARTICLE

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

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

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

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

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

With best regards,

The Editor

LETTERS TO THE EDITOR

Dear Editor,

With reference to the article "Loading Segments From FMP Files" by Larry W. Smith in Issue 4, Volume III of the Communicator/1000, I would first like to thank Larry for publishing this very helpful article. However,I would also like to point out a couple of problems with the SLOAD routine described in the article.

On page 23, line 155 of the program, all users should change the variable BPLEN in the READF call, to the variable LENR. Without this change, the base page area is not correctly loaded. The corrected statement reads as follows:

```
155 CALL READF (DCB1, IER, INBF1, LENR, LEN, RECORD)
```

The second problem, which will be encountered by RTE-II and RTE-III users only, occurs in lines 130 and 131. These lines must be modified to remove the +1 references from each. The corrected lines should read as follows:

```
130  PRGLEN=INBF24-ADDRES
131  BPLEN=INBF26-INBF25
```

Note that lines 130 and 131 should not be changed for RTE-IV and RTE-IVB users.

With these changes, the routines work extremely well on our RTE-II system, and have already saved us much anguish in working with segmented programs.

Again, thanks go to Larry Smith for this fine article.

Sincerely,

David W. Palmerston
Administrative Assistant
Helical Products Co., Inc.
Santa Maria, CA. 93456

Dear Mr. Palmerston,

As Editor of the Communicator, it is always gratifying to find out that one of the published articles has saved a reader from "anguish" of some kind.

Coincidently, at about the same time I recieved your letter, Larry also pointed out that the SLOAD routine was not quite correct as printed.

I apologize for any inconvenience this error may have caused you, or any other readers, and thank you for taking the time to report both the errors and the solutions back to me.

Sincerely,

The Editor

USER'S QUEUE

Dear Editor,

As one who has developed, and used for several years, a terminal handling program similar in function to the one described by Frank Fulton in "Scheduling Data Acquisition Programs Without Providing General System Access to Terminal Users" (Volume III, Issue 3), I found the article quite interesting. However, there appear to be a couple of errors in the listings which could cause the unsuspecting user some problems.

In the outline of the schedule program, (SCHED on page 28) the statement which disarms the terminal should be:

```
CALL EXEC(3,2100B+LU)
```

Likewise, the statement which re-arms the terminal should be:

```
CALL EXEC(3,2000B+LU)
```

Similar corrections should be made on page 29.

Finally, the re-arm statement on page 31 should also be:

```
CALL EXEC(3,2000B+LU)
```

As another suggestion, the schedule program and duplicator could be made significantly smaller (and thus assigned to a small partition, improving overall system response) by eliminating the formatter and read/write statements. These can be replaced with Data statements, and REIO calls respectively.

Hats off to Mr. Fulton for a well written article.

Very truly yours,

R. E. Hilton, Jr.
Applied Business Systems Corp.

Dear Mr. Hilton,

I am pleased that you found Mr. Fulton's article interesting, and I am also glad that you were able to take the time to point out these misprints.

Whenever possible, I try to actually compile, load, and run the code that appears in the Communicator/1000, to verify that all of the steps in the translation from a contributed article, to the final printed version of that article, have not caused any problems with the programs which are included. However, due to the nature of Mr. Fulton's article, and the equipment configuration that would be necessary to exercise his programs, I was not able to perform the verification as I usually do.

To you Mr. Hilton, I offer my thanks for pointing this error out to me and other Communicator readers. To those readers and Mr. Fulton, I offer my apologies in the hope that the errors in this article did not cause any insurmountable problems for readers that attemped to implement this scheme.

Sincerely,

The Editor

WHO OWNS THAT TYPE 6 FILE?

Clark Johnson/HP Data Systems Division

Running under RTE-IVB, one cause of some concern may be the management of space on LU's 2 & 3 for type 6 files. In order to manage this space well, I have found it useful to identify the owner of any type 6 files. Then, when more space is required on the cartridge, I can ask each owner about the significance of the type 6 files he has created. The program "WHOSE" was written to allow identification of type 6 files under RTE-IVB.

The heart of the program consists of two subroutines, "OID" and "UG". OID (Owner ID) opens the type 6 file and gets the 38th word (0 origin) which is the Owner ID number of the user that SP'd the program. UG (User-Group) takes the owner ID number and gets the corresponding user and group identifier strings from the accounts file. This information is returned to the main. The main (WHOSE) handles the user interface and the output.

The program is run with the following string:

```
RU,WHOSE,filename[,lu]
```

where: "filename" is the name of the type 6 file in question
"lu" is the output LU.

For example, the run string:

```
RU,WHOSE,WHOSE
```

results in:

```
File: WHOSE --> USER.GROUP = CLARK . EJ
```

indicating that the type 6 file " WHOSE " was SP'd by user CLARK belonging to the group EJ.

Or, as a second example, the run string:

```
RU,WHOSE,TRASH
```

may result in:

```
File: TRASH -->USER.GROUP = USER . UNKNOWN
```

WHOSE may also be run from a transfer file to produce a list of several type 6 files and their owners.

Given below are the main (WHOSE), the owner-ID subroutine (OID), and the user-group subroutine (UG).

```
FTN4,L
      PROGRAM WHOSE(), Main for OWNER ID info on type 6 files
      IMPLICIT INTEGER (A-Z)
C  ****************************************************************
C  *  This program finds the owner of a type 6 file.              *
C  *  The program is run by the string;   RU,WHOSE,file[,lu]      *
C  *  where lu is optional. The output will go to lu (and the     *
C  *  default is the user's terminal).                            *
C  *                                                              *
C  *  Date: 12-26-79                                              *
C  *  Programmer: Clark Johnson                                   *
C  ****************************************************************
```

```
      DIMENSION DCB (144),                                FILE (3)
      DIMENSION USER (5),                                 GROUP (5)
      DIMENSION INPUT (20),                               PARSD (33)
C                   get the file name
      EQUIVALENCE (PARSD (2),FILE)
C                   get the lu that was passed
      EQUIVALENCE (PARSD (6), LU)
C         character for nice output    , bad file name message
      DATA DOT /1H./             ,USER /2HOW,2HNE,2HR /, ULEN /5/
C         maximum characters for GETST , bad file name message
      DATA MAX /-20/             ,GROUP /2HUN,2HKN,2HOW,2HN /
      DATA GLEN /7/
C*** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
C                   THE PROGRAM
C                   get the run string
      CALL GETST (INPUT,MAX,LEN)
      CALL PARSE (INPUT,LEN,PARSD)

C                   get the LU for output if not supplied
      LU = PARSD (6)
      IF (PARSD (5) .EQ. 0) LU = LOGLU (J)

C               If no file name supplied, ask for it
      IF (PARSD (1) .NE. 0) GO TO 111
           WRITE (LU,100)
 100       FORMAT ("THE FILE NAME? ")

           READ (LU,101) FILE
 101       FORMAT (3A2)
 111  CONTINUE
C                   If ?? supply some information
      IF (PARSD (2) .NE. 2H??) GO TO 112
      WRITE (LU,113)
 113  FORMAT (/,
     &"Use the run string",/,"  :RU,WHOSE,filename [,output lu] "
     &,/," to identify the owner of a type 6 file.")

      GO TO 999
 112  CONTINUE

      IF ( OID (FILE,ID)  .EQ. 0) GO TO 200

      CALL UG (ID,USER,ULEN,GROUP,GLEN)

C                   The output
 200  CONTINUE
      WRITE (LU,401)FILE,(USER(J),J=1,(ULEN+1)/2),DOT
     &,(GROUP (I),I=1,(GLEN+1)/2)
 401  FORMAT (" File: ",3A2," --> "," USER.GROUP = ",11A2)

      GO TO 999
 990  CONTINUE
      STOP 2
 999  CONTINUE
      END
```

```
FTN4,L
      FUNCTION OID (FILE,ID)

C     This function takes name of a type 6 file, and returns
C     the 38th word (0 origin) in ID, and as the value of the
C     function. Under RTE-IVB, this word is the owner ID number.
C     If there is some error, ID and the function are set to 0.

      IMPLICIT INTEGER (A-Z)
      DIMENSION DCB (144),          RECORD (128)        ,FILE (3)
      EQUIVALENCE (RECORD (39),OWNID)

      DATA RLEN /128/              ,NOEXCL /1/          ,TYPONE/4/
      DATA CRN /-2/                ,SC /0/

      OID = 0
      ID = 0
      OPTION = NOEXCL + TYPONE

C           Open the file as type 1 on LU 2 (if no luck, try LU 3)
 100  CALL OPEN (DCB,ERROR,FILE,OPTION,SC,CRN)
           IF (ERROR .EQ. 1) GO TO 200

C                    Its not on LU 2? Then try LU 3
      IF (CRN .EQ. -3) RETURN
C     ELSE
           CRN = -3
           GO TO 100

C                    Read the first 128 word record

 200  CALL READF (DCB,ERROR,RECORD,RLEN,LEN)

           IF (ERROR .LT. 0) GO TO 990
           IF (LEN .LT. 38) GO TO 990

C                    Set the function value
      OID = OWNID
      ID = OWNID
      CALL CLOSE (DCB)
      RETURN

990   CONTINUE
C                    Handle the READF errors (ID and OID already 0)

      CALL CLOSE (DCB)
      RETURN
      END
```

```
FTN4,L
      SUBROUTINE UG (ID,USER,ULEN,GROUP,GLEN)
      IMPLICIT INTEGER (A-Z)

C     This subroutine will take a user ID as input and return the
C     USER and GROUP identifier strings.
C     The array USER will be of length ULEN, and GROUP will be
C     GLEN long.

      DIMENSION ACCT (3),          ADCB (144)
      DIMENSION RECORD (128),      ENTRY (16,8)
      DIMENSION USER (5),          GROUP (5)

      EQUIVALENCE (RECORD,ENTRY)
C                  account file name       record length
      DATA ACCT /2H+,2HCC,2HT!/           ,RLEN /128/
C        security code     cartridge reference number of accts file
      DATA SC /-31178/                    ,CRN /-2/
C                  Masks for left and right bytes
      DATA LEFT /177400B/                 ,RIGHT /377B/

C                  open the accounts file
      CALL OPENF (ADCB,ERROR,ACCT,1,SC,CRN)
           IF (ERROR .NE. 1) GO TO 990

C                  Get the first record
      CALL READF (ADCB,ERROR,RECORD,RLEN,LEN)
           IF (ERROR .LT. 0) GO TO 990
           IF (LEN .EQ. -1) GO TO 990

C                  record number for start of directory
      DREC = RECORD (5)
C                  record number for start of accounts info
      AREC = RECORD (6)

C                  Find the record that has the correct id in it
C     so first:    get the first record of the directory section

      DO 5 N = 2,DREC-1
           CALL READF (ADCB,ERROR,RECORD,RLEN,LEN)
                IF (ERROR .LT. 0) GO TO 990
                IF (LEN .EQ. -1) GO TO 990
 5    CONTINUE
 100  CONTINUE

           CALL READF (ADCB,ERROR,RECORD,RLEN,LEN)
                IF (ERROR .LT. 0) GO TO 990
                IF (LEN .EQ. -1) GO TO 990

C                  then, look for the correct ID number
           DO 10 N = 1,8
                RNUM = N
                IF (ENTRY (12,N) .EQ. ID) GO TO 700
 10        CONTINUE

C                  couldn't find it in that record, try the next?
           DREC = DREC + 1
           IF (AREC .GT. DREC) GO TO 100
```

```
C                    Oh well, we failed so give up
      GO TO 900
C                    The correct user has been found
 700  CONTINUE

      DO 20 N = 1,5
           USER (N) = ENTRY (N+1,RNUM)
           GROUP (N) = ENTRY (N+6,RNUM)
 20   CONTINUE

C                    get the user and group string length

      GLEN = ENTRY (1,RNUM) .AND. RIGHT
      ULEN = (ENTRY (1,RNUM) .AND. LEFT) /256

 900  CALL CLOSE (ADCB)
      RETURN
C                    A file failure somewhere
 990  ULEN = -1
      GO TO 900
      END
```

CRT SCREEN ENHANCEMENTS

Alan Tibbetts/HP Neely Santa Clara

A little known trick in RTE which will make interactive dialogue look a lot better is to put an underscore (back-arrow on older terminals such as TTY's) character as the last character in a line of output. The RTE drivers for terminals, DVR05, DVA05, and DVR07 interpret this to mean that they should suppress the CRLF which is normally output at the end of each line. This means that the cursor will stay right on the same line where you have just asked the user a question. Although this feature is documented in the driver manuals, it seems that most programmers do not know about it.

Trick Number 2 — If you would like to have a counter or some other status which updates continuously, but you don't want the CRT to scroll up and lose all your printing, there is a way you can write over the same line continuously. An example is the best way to show this:

```
FTN4,L
        PROGRAM EXMPL
        DATA NOLF/6537B/
        DO 100 I = 1, 20
        WRITE (1,1) I, NOLF
1       FORMAT ("I=" I2,A2)
100     CONTINUE
        END
```

When you load this program and run it, you will see a counter which will write 20 times on one line at the bottom of your CRT. The reason for this strange behavior is that the constant NOLF is, in the left byte a carriage return, and in the right byte an underscore. The carriage return causes the cursor to go to the left side of the screen, and the underscore inhibits the CRLF which the driver would normally output, which would make the cursor go to the next line.

Notice that this does not require any particular brand of CRT (such as H-P).

WARNING! THIS WILL NOT WORK WITH LINE PRINTERS.

HOW TO PREVENT DS/1000 MONITORS FROM HOGGING PARTITIONS

Lyle Weiman/Data Systems Division

In a recent issue of Support Update, ("Don't Let DS Monitors Hog Your Partitions!"), Harvey Bernard stated that the way to prevent DS/1000 monitors from tying up partitions, was to assign them all to the same one or two partitions. Harvey's reasoning is absolutely correct, but partition hogging can occur with more than just the DS monitors.

When the RTE dispatcher begins looking for a partition in which to run a program that has just become scheduled, it first looks for an empty partition. If none are available, the dispatcher looks for the smallest partition whose occupant has a priority lower than the newly-scheduled program. (This occupant must also be swappable, of course.) The key point here is that DS monitors are supplied to run at priority 30, and there can be up to 15 of them in a node that contains all of the "bells and whistles". Thus, these programs could tie up as many as 15 partitions, forcing all programs of lower priority to swap in and out of the partitions that remain. If there are fewer than 17 partitions, your system will perform as though there was only one swapping area.

Mr. Bernard's suggestion, that DS monitors be assigned to one or two partitions, will certainly solve this particular problem, but ignores the general case: RTE has no way of removing high-priority programs which aren't being used. It should be clear that this curse affects all programs which spend a good deal of time in swappable "wait" states. We need a solution for this general case.

It would also be nice if all the DS monitors did not have to compete with one another for partition space. Thus, during periods when several monitors are required, response time and throughput would not suffer as a result of constant swapping.

There is a solution to the general problem. We have used the program described below for over a year to sucessfully "free-up" partition space.

The program FLUSH, runs in the time list, at intervals set by the user. Its job is to scan the memory allocation tables, and force a swap of any program which is not in need of that partition at that moment, that is, programs which are suspended in the general-purpose wait state (3). To force a swap, FLUSH requires the program SWAPT, a program which terminates immediately upon being executed. FLUSH assigns SWAPT to the partition to be cleaned out, via a call to the system message processor, and the ASSIGN command (for this reason, the programs can only be used in RTE-IVA/B). FLUSH then schedules SWAPT, without wait, and re-schedules itself to run one second later. This forces the occupant of that partition to be swapped out. SWAPT terminates immediately upon being executed, thus returning the partition to "free" status. If, for some reason, SWAPT could not be "rolled" into that partition, FLUSH will terminate SWAPT as soon as FLUSH is re-scheduled, one second later. This safety feature prevents any problems that could result in case events should change between the time when FLUSH makes its determination to force a swap, and when the program SWAPT is scheduled. FLUSH repeats this sequence for all of the partitions in the system, and then terminates itself.

To force a swap, the following conditions must all be met:

1. The partition must be occupied.
2. The occupant must be in state 3 (general wait).
3. The occupant must not have set the "do not swap me" bit.
4. The occupant must not be an EMA program.

You may wish to amend the above list, according to whatever rules make sense at your installation.

One final note, the one second interval that FLUSH uses, (explained above) destroys its interval time, so you can't use the IT command to set the interval. The interval is passed as a parameter, and defaulted to one minute. For this reason, FLUSH never really terminates, it just offset-schedules itself, and hence it ties up one partition. If this is not acceptable, an alternative would be to lower its priority, so that when the system is quite busy, FLUSH doesn't waste time swapping programs. Using these procedures, your system can be "custom-tuned" for peak performance.

```
ASMB,R,Q
      NAM FLUSH,2,30 8-23-79 -LAW-
*
*     PROGRAM TO FLUSH HIGH-PRIORITY PROGRAMS OUT OF PARTITIONS
*     AND KEEP SWAPPING TO A MINIMUM.  THE PROBLEM IS THAT RTE
*     WILL LOAD UP ALL PARTITIONS WITH PROGRAMS AS SOON AS THEY
*     ARE SCHEDULED.  IF THEIR PRIORITY IS HIGH AND THEY DO NOT
*     TERMINATE, THEY MAY BECOME SWAPPABLE, BUT REMAIN IN THAT
*     PARTITION DESPITE THEIR INACTIVITY.  THIS CAUSES RTE TO UTILIZE
*     ONLY ONE PARTITION FOR SWAPPING, WITH CONSEQUENT HIGH OVERHEAD.
*     THE PROBLEM IS WORSENED BY DISTRIBUTED SYSTEMS SOFTWARE, WITH
*     MANY PROGRAMS WHICH ARE INFREQUENTLY USED OCCUPYING PARTITION
*     SPACE.  THIS PROGRAM FORCES THEM OUT OF THERE.
*
*     TO RUN:
*
*     *ON,FLUSH[,<INTERVAL>]
*
*     WHERE <INTERVAL> = PERIOD OF TIME (IN MINUTES) TO WAIT
*                        BETWEEN ``FLUSHINGS''.
*
*
          EXT $MATA,$MNP
          EXT RMPAR,CNUMD,EXEC,MESSS,.MVW
*
*     SET UP LOOP TO SCAN EACH PARTITION, LOOKING FOR
*     FOR ONE WITH AN OCCUPANT WHICH SHOULD BE SWAPPED.
*     CONDITIONS ARE:
*     1. PARTITION IS OCCUPIED.
*     2. OCCUPANT IS STATE 3
*     3. OCCUPANT HAS NOT SET THE ``DO NOT SWAP ME'' BIT
*     4. OCCUPANT IS NOT AN EMA PROGRAM
*
FLUSH EQU *
      JSB RMPAR      RECOVER PARAMETERS
      DEF *+2
      DEF MSGF2
      LDA MSGF2      LOAD INTERVAL
      SSA,RSS        ONE SPECIFIED?
      CCA            NO, USE ONE MINUTE
      STA INTRV      SAVE INTERVAL
```

```
*
START EQU *
      LDA $MNP
      CMA,INA
      STA CNTR
      LDA $MATA      SETPOINTER TO $MATA TABLE
      ADA D2         INCREMENT TO ID SEGMENT WORD
      STA PNTR
      CLA,INA
      STA NPART      INITIALIZE FOR PARTITION   1
LOOP  EQU *
      XLB PNTR,I     LOAD ID SEGMENT OF OCCUPANT, THIS PARTN
      SZB,RSS        ANY OCCUPANT?
      JMP NEXT       NO, GO ON TO NEXT ONE
      CPB XEQT       IS THIS ME?
      JMP NEXT       YES, DON'T DO IT!
      ADB =D14       ADVANCE TO MEMORY-LOCK
      XLA B,I        LOAD WORD 14
      AND MLOCK      CHECKFOR MEMORY-LOCK
      SZA               CAN THIS GUY BE SWAPPED?
      JMP NEXT       NO
      INB            ADVANCE TO STATUS WORD
      XLA B,I        LOAD THE
      AND =B17          PROGRAM'S STATUS
      SZA,RSS        DORMANT?
      JMP SWHIM       --YES, SWAP HIM
      CPA D3         STATE 3?
      RSS            YES, CONTINUE
      JMP NEXT       NO, DON'T BUMP THIS GUY
      ADB =D13       ADVANCE TO WORD 28
      XLA B,I        IS THIS
      SZA              PROGRAM USING EMA?
      JMP NEXT       YES, DON'T SWAP IT
*
*     THE PROGRAM IN THE PARTITION NUMBER GIVEN BY ''NPART''
*     SATISFIES OUR CONDITIONS TO FORCE A SWAP.
*
SWHIM EQU *          HERE TO FORCE OCCUPANT OF PRTN TO BE
*                    SWAPPED OUT.
      JSB CNUMD      CONVERT PARTITION NUMBER TO ASCII
      DEF *+13
      DEF NPART
      DEF MSGF1+5
      LDA MSF1      LOAD SOURCE ADDRESS
      LDB MSF2      LOAD DESTINATION ADDRESS
      JSB .MVW       MOVE ''AS,SWAPT,<PART >'' TO ANOTHER BUFFER
      DEF D8
D0    NOP
*                    ASSIGN ''SWAPT'' TO A PARTITION, THEN SCHEDULE IT.
*                    IF CURRENT OCCUPANT OF THAT PARTITION IS NOT SWAPPABLE,
*                    THEN IT CAN'T BE SCHEDULED, SO TERMINATE IT.
```

```
*
      JSB MESSS
      DEF *+3
      DEF MSGF2
      DEF D16
*                   NOW SCHEDULE THE ''SWAP-IT'' PROGRAM, FORCING OCCUPANT
*                   TO BE SWAPPED OUT.
      JSB EXEC
      DEF *+3
      DEF K10N
      DEF SWAPT
      JMP EXIT      SOMETHING WENT WRONG WITH THE SCHEDULE--QUIT.
*                   WAIT FOR ONE SECOND, SO SWAP CAN OCCUR
      JSB EXEC
      DEF *+6
      DEF D12
      DEF D0
      DEF D2        UNIT: SECONDS
      DEF D0
      DEF M1          FOR ONE SECOND
*
*     IN CASE THE SWAP WAS BLOCKED, WE NOW FORCEFULLY
*     ABORT THE ''SWAP-IT'' PROGRAM.
*
      JSB EXEC
      DEF *+4
      DEF K6N
      DEF SWAPT
      DEF D0
      NOP           DON'T CARE ABOUT ERRORS.
*
NEXT  EQU *          HERE TO ADVANCE TO NEXT PARTITION
      ISZ NPART     BUMP PARTITION NUMBER
*
      LDA PNTR
      ADA MATAS     ADD $MATA SIZE
      STA PNTR
      ISZ CNTR      BUMP LOOP COUNTER
      JMP LOOP      CONTINUE SEARCH
*
*     END OF THE LINE.  WAIT A BIT & START OVER
*
EXIT  EQU *
      JSB EXEC
      DEF *+6
      DEF D12
      DEF D0
      DEF D3
      DEF D0
      DEF INTRV
```

```
*
       JMP START      START UP AGAIN.
       SPC 2
*      DATA AREA
       SPC 2
A      EQU 0
B      EQU 1
XEQT   EQU 1717B      CURRENTLY-EXECUTING PROGRAM ID SEGMENT
MLOCK  OCT 100        ''MEMORY-LOCK'' BIT  ASK
M1     DEC -1
D2     DEC 2
D3     DEC 3
D6     DEC 6
D8     DEC 8
K6N    OCT 100006     NO-ABORT, PROGRAM TERMINATE
K10N   OCT 100012     NO-ABORT
D12    DEC 12
D16    DEC 16
MATAS  DEC 7          SIZE OF $MATA TABLE ENTRIES
PNTR   NOP
INTRV  NOP            INTERVAL TO WAIT BETWEEN RUNS.
CNTR   NOP
                                      NPART NOP
SWAPT  ASC 3,SWAPT    ''SWAP IT'' PROGRAM NAME
@MSF1  DEF MSGF1
@MSF2  DEF MSGF2
MSGF1  ASC 10,AS,SWAPT,
MSGF2  BSS 10
       END FLUSH
FTN4,L
       PROGRAM SWAPT(2,1),5-24-79
       END
       END$
```

USE OF THE 264X TERMINAL AS A DATA RECORDER

C. L. Elliot and J. L. McLaughlin/
ELLIOT GEOPHYSICAL COMPANY

ABSTRACT

Versatile features of the 264X terminal make it suitable for off-line digital data recording. This data can be rapidly processed by an HP 1000 computer system. By simple data manipulation, 12 bit digital words can be converted to two printing ASCII characters, which can be readily accepted by a 264X terminal, and stored on mini-cartridges. This is an economical way to remotely store data from X-Y digitizers and laboratory electronic instruments.

INTRODUCTION

How many times have you wished you had a simple but positive means around your office or laboratory, to input digital data on an HP 1000 from such instruments as X-Y digitizers, A/D converters, or laboratory electronic instruments with binary or BCD data outputs? It usually is a very simple matter if you have a standard compliment of HP I/O boards capable of serial or parallel data input, that are not dedicated to other uses. Data entry rates also may be of concern in that some laboratory instruments, such as X-Y digitizers produce data at such low data rates that it would not be economical to directly interface such instruments to the HP/1000. If you don't have such I/O boards, or are not willing to make a direct interface to the HP 1000, how can digital data be conveniently input? The answer is simple, make your 264X terminal a digital recorder. Not only can such a terminal be readily adapted for digital data recording, but it can be utilized in off-line operations that may be physically removed from the computer area. In an off-line mode, the 264X terminal allows slow data rate input, such as that usually obtained from a digitizer, which would otherwise prohibit direct line interfacing to the HP 1000.

Data stored by a HP 264X terminal, or its equivalent, is obviously readily interfacable to the HP 1000. An HP 264X terminal, or its equivalent, can hold many thousands of ASCII characters in its display memory with an appriopriate compliment of memory boards. Such data can then easily be dumped by means of a CTU to mini-cartridges, each of which will hold more than 100,000 ASCII characters.

IMPLEMENTATION

In our principal application using this technique, we desired to interface an X-Y digitizer with a 12 bit binary output, to our HP 1000 system. We chose to use an HP 264X Terminal to work off-line and remote from the computer as the intermediary data storage recorder. A decided plus was having available the terminal keyboard for annotation of the recorded data sets. Obviously, the 12 bit binary words cannot be directly input to an ASCII character terminal such as the 264X type. However, by a simple technique of massaging each binary word in hardware, it was possible to fool the terminal into believing it was receiving normal ASCII characters. Therefore the terminal stored them in display memory without any problem. It was then a simple matter to record the display memory data on mini-cartridges.

The general scheme of our X-Y digitizer interface system is shown in the following figure:

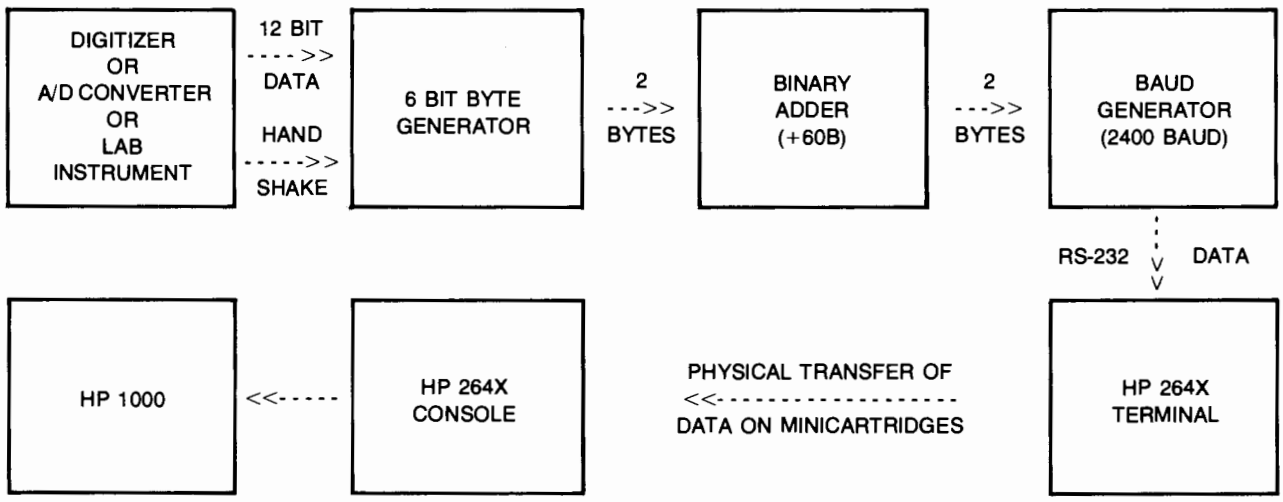


Figure 1. Flow Diagram

HARDWARE

Each 12 bit output word from the X-Y digitizer was hardware toggled into 6 bit data bytes representing the high order and low order bits of the original 12 bit word. In order to prevent ASCII control characters (00B to 37B) from interfering with normal terminal operation, each of the two 6 bit data bytes from the original data word was translated upward by 60B, thereby creating a normal printing character data set that would encompass the entire range of interest from binary 000000 to 111111 or 00 to 77 octal. This operation was carried out by means of a binary adder. The output of the binary adder, consisting of two shifted octal data bytes, was presented to a parallel-to-serial baud generator operating at 2400 baud. The output of the baud generator was a conventional RS232C serial data stream, consisting of valid ASCII characters.

The HP 264X terminal, acting as a data recorder, easily accepted this data stream, and stored it in display memory. It was then recorded, in blocks on a mini-cartridge, by means of the terminal CTU. The recorded mini-cartridges were physically carried to the computer area, re-read by means of the CTU on the HP 1000 console, and the data processed with program DIGIT. The following table illustrates the translation from 12 bit binary words, to shifted ASCII characters (shifted by 60B) that are stored by the data terminal recorder.

Table 1

INPUT DATA 12 BIT BINARY	OCTAL EQUIVALENT	SHIFTED OCTAL BYTES LEFT	SHIFTED OCTAL BYTES RIGHT	SHIFTED ASCII CHARACTERS
000000000000	0000	60	60	00
000000000001	0001	60	61	01
100000000000	4000	120	60	P0
101110100101	5645	136	125	^U
111111111110	7776	157	156	on
111111111111	7777	157	157	oo

SOFTWARE

An HP 1000 routine written in Fortran IV, named DIGIT, has been put together to implement this method of data recording by means of a 264X Terminal as reported in this article. The coding for this routine follows:

```
      PROGRAM DIGIT
C     UTILIZATION OF 264X TERMINALS AS 12 BIT BINARY DATA RECORDERS
C     ID = IDENTIFICATION LABEL
C     IDATA = INTEGER(12 BIT) OF INPUT DATA
C     DATA = REAL ARRAY OF SCALED INPUT DATA
C     NO = NUMBER OF DATA POINTS IN IDATA
      DIMENSION LINK(5),DATA(40),ID(20),IDATA(40)
      LU=LOGLU(LU)
      CALL RMPAR(LINK)
      IN=LINK(1)
      IO=LINK(2)
   10 ISTOP=0
      READ(IN,1000)(ID(I),I=1,40)
 1000 FORMAT(40A2)
C     TEST FOR EOF = **
      IF(ID(1).EQ.2H**)GOTO9999
      WRITE(LU,1000)(ID(I),I=1,40)
C     FOLLOWING SCALING FACTORS DETERMINED BY APPLICATION
      WRITE(LU,1500)
 1500 FORMAT(" DECIMAL SPAN AND OFFSET FOR 12 BIT BINARY_? ")
      READ(LU,*)SPAN,OFSET
      WRITE(IO,1000)(ID(I),I=1,40)
   15 CALL ASCTB(NO,IDATA,IN,LU,ISTOP,NCHR)
      IF(ISTOP.EQ.1)GOTO9999
      IF(ISTOP.EQ.2)GOTO10
      CALL OTOFP(NO,IDATA,DATA,LU,SPAN,OFSET)
      WRITE(IO,2400)(DATA(I),I=1,NO)
 2400 FORMAT(10F8.3)
      IF(NCHR.LT.80)GOTO10
      GOTO15
 9999 END
      SUBROUTINE ASCTB(NO,IDATA,IN,LU,ISTOP,NCHR)
C     READ IN SHIFTED ASCII CHARACTERS EACH CONTAINING 12 BIT DATA BYTES
C     TWO ASCII CHARACTERS = ONE 12 BIT DATA WORD
C     IL = LEFT CHARACTER = MOST SIGNIFICANT 6 BIT BYTE
C     IR = RIGHT CHARACTER = LEAST SIGNIFICANT 6 BIT BYTE
C     SHIFT = 60B TO FORM PRINTING CHARACT. FOR PROCESSING BY 264X TERMINAL
C     E.G.,000000 = 0, 000001 = 1,- - - - 111111 = o (LOWER CASE LETTER O)
C     NO = NUMBER OF DATA WORDS IN IDATA
C     IDATA = INTEGER DATA ARRAY
C     END OF DATA SET = TWO CONSECUTIVE DECIMAL POINTS
C     EOF = ** IN COL. 1 & 2
      DIMENSION IBUF(80),JBUF(40),IDATA(1)
      ICNWD=IOR(400B,IN)
      CALL EXEC(1,ICNWD,JBUF,-80)
      CALL ABREG(IA,NCHR)
      IF(NCHR.EQ.0)GOTO975
      DO 50 I=1,NCHR,2
      II=I+1
      IBUF(I)=2H
      IBUF(II)=2H
```

```
C      SGET = HP DECIMAL STRING ARITHMETIC ROUTINE
       CALL SGET(JBUF,I,IBUF(I))
       CALL SGET(JBUF,II,IBUF(II))
   50  CONTINUE
       DO 100 I=1,NCHR
C      TEST FOR END OF DATA SET
       IF(IBUF(I).EQ.56B)GOTO200
C      TEST FOR INVALID DATA
  110  IF(IBUF(I).GT.157B)GOTO 900
       IF(IBUF(I).LT.60B)GOTO 900
       GOTO100
  200  IF(IBUF(I+1).NE.56B)GOTO110
       NCHR=I-1
       GOTO250
  100  CONTINUE
  250  ITEST=MOD(NCHR,2)
       IF(NCHR.EQ.0)GOTO975
C      TEST FOR COMPLETE DATA (EVENLY DIVISIBLE BY 2)
       IF(ITEST.NE.0)GOTO950
C      REGENERATE UNSHIFTED 12 BIT BINARY WORDS IN IDATA ARRAY
       M=0
       DO 500 NL=1,NCHR,2
       NR=NL+1
       M=M+1
       IL=(IBUF(NL)-60B)*64
       IR=IBUF(NR)-60B
       IDATA(M)=IOR(IL,IR)
  500  CONTINUE
       NO=M
       RETURN
C      ERROR MESSAGES WRITTEN TO LU
  900  ICOL=I
       WRITE(LU,2100)ICOL,(IBUF(I),I=1,NCHR)
 2100  FORMAT(1H0,////,10X,"BAD CHARACTER IN THE FOLLOWING LINE AT",
      $" COLUMN = ",I2," ---- EXECUTION TERMINATED",//,10X,80R1)
       ISTOP=1
       RETURN
  950  WRITE(LU,2200)(IBUF(I),I=1,NCHR)
 2200  FORMAT(1H0,////,10X,"CHARACTER STRING IS NOT AN EVEN",
      $" MULTIPLE OF TWO",//,10X,80R1,//,"EXECUTION TERMINATED")
       ISTOP=1
       RETURN
  975  ISTOP=2
       RETURN
       END
       SUBROUTINE OTOFP(N,NIN,XOUT,LU,SPAN,OFSET)
C      OCTAL TO FLOATING POINT CONVERSION
       DIMENSION NIN(1),XOUT(1)
       FACT=SPAN/4096.0
       DO 100 I=1,N
       IF(NIN(I).EQ.0.OR.NIN(I).EQ.7777B)GOTO 200
       XOUT(I)=FACT*FLOAT(NIN(I))+OFSET
       GOTO 100
C      JAM OUTPUT TO 999.0 FOR UNDERFLOW OR OVERFLOW
  200  XOUT(I)=999.0
  100  CONTINUE
       RETURN
       END
       END$
```

The program DIGIT, with subroutines ASCTB and OTOFP, perform the necessary software decoding of the shifted ASCII characters. Means are available to input decimal "span" and "offset" parameters to scale the decimal data words from 0000B to 7777B. These parameters are input by means of the operating LU. Data input is by means of mini-cartridge, and output is operator selected. Subroutine ASCTB converts shifted ASCII characters, each containing 6 bit data bytes, back to normal unshifted 12 bit binary data words. Subroutine OTOFP converts these binary data words to floating point data with appropriate offset and span adjustments to the selected arbitrary zero reference. Underflow or overflow data is trapped in this subroutine, and the corresponding word in the output is set to 999.000 to indicate an underflow or overflow condition.

During recording of data sets, an end-of-data indicator is inserted by means of the 264X terminal keyboard, by two consecutive periods. The period (.), has an ASCII representation of 56B, and therefore is not a legal character for an instrument to attempt to send. The period cannot be confused with valid data, and is easily tested for, in the subroutine ASCTB. An end-of-file mark during data recording is inserted by means of the keyboard, by two consecutive asterisks (**) on the final line of the file, in columns 1 and 2. The asterisk is an ASCII character represented by 52B, and therefore is also an invalid data character. The EOF mark is trapped in the main program DIGIT.

EXAMPLE

A test data set is now presented as a further means to illustrate the shifted character recording technique. For this data, a span of 10.0 and an offset of 1.0 has been arbitrarily used. These values are input by means of the operating console when requested by the program DIGIT. The shifted ASCII character record on the minicartridge is as follows:

```
 TEST DATA FOR PROGRAM DIGIT       SPAN = 10.0     OFFSET = 1.0
\c62;d6:5e6=5e6Bloo1k6g5d3[ee=cSdgnGd7g:e7CLd\OEd7<;:7b>e7h=d7m5c81
 cb15c80]cm2
=<klnn5c=S][==;b=o5c..
**
```

The first line is title information in which span and offset values have been recorded for later input through the LU. Two decimal points terminate the data stream on the third line, and two asterisks on line 4 indicate end-of-file. This data stream contains 50 data points.

During execution of program DIGIT, an octal dump of the integer data array IDATA in subroutine ASCTB would appear as follows:

```
5463  0602  1364  0612  0565  0615  0565  0622  7420  7777
0173  0667  0564  0353  6565  1563  4364  6776  2764  0767
1265  0723  3464  5437  2564  0714  1312  0762  1665  0770
1564  0775  0563  1001  5363  6201  0563  1000  5563  7502
1514  7374  7676  0563  1543  5553  1515  1362  1577  0563
```

Finally, the resulting output at LU=IO from execution of program DIGIT is as follows. (This is the normal output of the recovered decimal data.)

```
TEST DATA FOR PROGRAM DIGIT       SPAN = 10.0     OFFSET = 1.0
8.000    1.942    2.846    1.962    1.911    1.969    1.911    1.981   10.414 999.0
1.300    2.072    1.908    1.574    9.411    3.156    6.596    9.745    4.721    2.2
2.692    2.140    5.502    7.951    4.408    2.123    2.743    2.216    3.317    2.2
3.158    2.243    1.906    2.252    7.843    8.815    1.906    2.250    8.156   10.5
3.061   10.365   10.839    1.906    3.117    8.136    3.063    2.841    3.185    1.9
```

The following table illustrates the data paths from decimal input data to recovered data, for several of the selected data words from the test data set. Of special note is data word No. 10 which is an overflow condition. For this case the software sets the output to 999.000 so that in subsequent analysis of the data, for whatever purpose, underflow or overflow data can be easily trapped by means of appropriate software.

Table 2

WORD NO.	DECIMAL INPUT DATA	SHIFTED OCTAL	SHIFTED BYTES	SHIFTED ASCII CHARACTERS	RECOVERED OCTAL	RECOVERED BYTES	RECOVERED DATA
1	8.000	134	143	/c	54	63	8.000
2	1.942	66	62	62	06	02	1.942
3	2.846	73	144	;d	13	64	2.846
4	1.962	66	72	6:	06	12	1.962
10	(Overflow)	157	157	00	77	77	(Jammed to 999.000)

We have found this system of utilizing a 264X data terminal as a remote recorder, a convenient, straightforward, and economical means for obtaining data not only from an X-Y digitizer, but other laboratory instruments as well. Resulting data is conveniently input to an HP 1000 for processing. With a little special programming, the user can take full advantage of some of the versatile features of the 264X series of terminals in storing and transmitting data from the point of generation to an HP 1000, for rapid and convenient processing.

AN OVERVIEW OF THE PASCAL LANGUAGE

Van Diehl/HP Data Systems Division

[Editor's Note: This is the first of a two part series by Van on the PASCAL language. The first part, which appears below, is intended as an general overview of the language. The second part of the article, which is to be published in a future issue of the Communicator, will deal explicitly with HP's new PASCAL compiler.]

INTRODUCTION

Have you ever considered building a mechanical assembly or designing a building, using beams, bolts, screws, etc. out of non-standard structural components? Or, have you ever considered designing a process or building, starting not with the objectives of the final design, but with the design of basic components? If such design methods look ridiculous and would never be considered in most engineering design processes, they did not seem so ridiculous to a large population of engineers and scientists for the design of computer programs.

Perhaps because the science of computer programming is a relatively new one and does not have the large accumulated set of design constraints and rules imposed by the vast body of existing engineering standards, the design of computer programs was, and often is, rather unruly, and the design methods used change from programmer to programmer. The end result is that large computer programs are very difficult to make work, and when they finally work, are very expensive to maintain.

To improve the programming process and thus lower the cost of software development and maintenance, it was felt that more analytical programming techniques were needed. Some of the more important techniques that were introduced to bring a methodology to computer programming are structured programming, and top-down design or successive refinement. Structured design in a broader sense means writing programs using fewer building blocks and using building blocks that are sufficiently decoupled, thus avoiding undesirable side effects. The structured programming design method is a method to facilitate the writing of correct programs using building blocks, (programming components) that are decoupled, i.e., have one entry and one exit point. Not only do all procedures and subroutines have one entry and one exit, but every programmming component of the structure, down to the level of individual statements, obeys this general rule.

The advantages of a structured approach to the design of a program, as well as to the coding are many: programs can be made to work with less effort, testing is quicker, and "bugs" are more easily fixed because of the inherent independence and modularity of structured programs.

One important side effect of structuring a program is the increased readability of the resulting code. Since the control flow exhibits no major jumps and is generally linear, the text of the program can be read more like a normal text, without unusual jumps. The structured programming method is often associated with a top-down design approach.

Top-down design and successive refinement are methods of program design that start by specifying the program design in its more general terms, then expand these into more and more specific and detailed actions, until the whole program is complete.

The top-down hierarchical structure of a program is obtained by the use of procedures. A program will require high-level procedures to carry out large difficult tasks, such as solving a set of simultaneous equations. These procedures will call procedures at a lower level in the hierarchy, down to primitive procedures at the lowest level such as 'read a character'. The higher-level procedures will express the general and abstract features of the problem solutions and the low-level procedures will contain the details of implementation. Writing a program using a top-down design or step-wise refinement will consist of proposing a very abstract solution and then successfully refining it until it is entirely expressed in the chosen programming language.

The structuring of programs does not stop in the decomposition of the problem in procedures. There are three other basic structures that the procedure can be decomposed into: the sequence, the decision, and the repetitive structure or loop. The sequence is a group of instructions executed one after the other. The decision is a section of the program which is influenced by the data. The structure is used to execute an instruction or sequence of instructions several times.

Each program is composed of the actions which are to be performed, and a description of the data which are manipulated by the language statements. These two elements are combined in structured building blocks, which specify specify sequented, selective or repeated execution of their components.

The data are described by data structuring statements, that specify the variables used and their types. The data type essentially defines the set of values which may be assumed by the variable. The expressive power of a language derives largely from its data types, its data structuring declaratives, and its primitive operations on data structures.

As we will later discuss, the programming language PASCAL uses all of these techniques of structuring. They are incorporated in a simple and elegant fashion which makes PASCAL a powerful language that is nonetheless easy to use.

THE PASCAL LANGUAGE

The languages that we use have a profound effect on the way we think. Thus, it is important to teach programmers, as early as possible, programming languages that stress the concepts of structured programming. PASCAL was designed specifically to aid in programming in a structured style. Thus writing structured programs in PASCAL is a straightforward matter of using the facilities made available to the programmer. The constructs are innately structured, forcing the programmer to write structured code; writing unstructured code is possible, but one is working against the nature of the language rather than with it.

Writing structured code in FORTRAN is the reverse situation. Here the structuring is working against the basic nature of the language. But, more important than the algorithmic structuring facilities of the PASCAL language, the expressive power of the language is derived from its data structuring facilities. PASCAL belongs to a new class of languages reflecting the importance of data management. In this regard PASCAL utilizes ideas and features of COBOL, PL/1, and ALGOL. COBOL, still the most widely used high-level language, was the first to provide data structure declarations. See table 1.

PASCAL incorporates most of the data facilities of COBOL and PL/1, but in an ALGOL-like framework. PASCAL certainly dwarfs FORTRAN in this area. PASCAL and FORTRAN include provisions for certain primitive data types, i.e., reals, integers, and logical variables. PASCAL also includes a primitive for characters. However, the range of structured data types is considerably more limited in FORTRAN than in PASCAL. FORTRAN allows only two structured data types — COMPLEX and ARRAYS. PASCAL on the other hand allows a limitless variety of data structures.

Table 1

HIGH LEVEL LANGUAGES HISTORICAL SUMMARY

LANGUAGE	DATE OF INTRODUCTION	APPLICATION
FORTRAN	1957	Numerically Oriented Language
ALGOL	1960	Also Numerically Oriented Language
COBOL	1960	Business Language
LISP	1961	List Processing and Symbolic Manipulation
SNOBOL	1962	Character String Processing
BASIC	1965	Interactive Programming Language
PL/1	1965	Complex, General Purpose Language for Numerical and Non-Numerical Applications
APL	1967	Interactive Language for Vector Manipulation
PASCAL	1971	General Purpose Language to Teach the Concents of Programming and Allow the Efficient Implementation of Large Programs. Descendant of ALGOGLO and ALGOWL

PASCAL PROGRAM STRUCTURE

A PASCAL program structure is similar to an ALGOL procedure. It begins with a program heading, is followed by a block, and is terminated by a period. The program heading contains the name of the program and a list of external file variables used by the program. These files are the means by which the program communicates with the outside world. The block contains a compound statement, i.e., a list of single statements bracketed by BEGIN and END. The block may be preceded by declarations.

Below is shown the skeleton of a PASCAL program.

```
PROGRAM  name (list of files);
LABEL    1,2;
CONST    (*constant definitions*)
         pi = 3.14;
         k = 5;
TYPE     (*data type definition*)
  .
  .
  .
VAR    (*variable declarations*)
  .
  .
  .
PROCEDURE    abc (parameter list);
```

BLOCK

```
PROCEDURE    xyz  (parameter list);
```

BLOCK

```
FUNCTION    X (parameters) : TYPE;
```

BLOCK

```
BEGIN
```

Compound Statement

```
END.
```


Let us write a simple PASCAL program to illustrate the top-down structure of the language. Our example program determines the first term of the Fibonacci series that is greater than 1000. The Fibonacci series is the following series:

0,1,1,2,3,5,8,13,21,34,...

In general the j^{th} term is found from its predecessor by the recurrence rule:

$$term_j = term_{j-1} + term_{j-2}$$

Let us implement the program using the variable "sum", to hold the value of the most recently generated term, $term_j$, while the variables "latest" and "next_latest", respectively, acquire values representing $term_{j-1}$ and $term_{j-2}$.

```
PROGRAM fibonacci;
    VAR
        next_latest, latest, sum: integer;
    BEGIN (*initialize*)
        BEGIN
              next_latest:=0;
              latest:=1;
              sum:=0
        END;

    WHILE sum<=1000 DO
    BEGIN
        sum:=sum+next-latest;
        next_latest:=latest;
        latest:=sum
    END;
        writeln (sum)
END.
```

The two examples in figures 1a and 1b show a section of program code written in FORTRAN and PASCAL that both perform a limit comparison in a steam plant startup program. Note that the variable names (identifiers) could have been chosen, in PASCAL, to convey more meaning than they do.

```
  C FORTRAN SOURCE CODE

    RPMBIT=0
    IF (TRT .GE. 14.1) GOTO 200
    HT 1300=0.0
    GOTO 110
200 IF (TRT .GT. 23.4) GOTO 118
    HT 1300=TRT -14.4
    GOTO 110
118 IF (TRT .GT. 28.4) GOTO 111
    HT 1300=9.0
    HT 2400=TRT -23.4
    GOTO 10
111 RPMBIT=1
    IF (TRT .GT. 32.4) GOTO 122
    HT 1300=TRT -20.4
    GOTO 110
122 HT 1300=.33 * TRT
    HT 2400= .67 *TRT -20.4
    GOTO 10
110 HT 2400=0.0
 10 CONTINUE
    END
```

Figure 1a

```
            (**PASCAL SOURCE CODE**)

TYPE
  TEST_RESULT=(1,2,3,4,5);
VAR
  TEST: TEST_RESULT;
       .
       .
       .
       IF TRT < =14.4 THEN TEST:=1
          ELSE IF TRT <=23.4 THEN TEST:=2
               ELSE IF TRT <=28.4 THEN TEST:=3
                    ELSE IF TRT <=32.4 THEN TEST:=4
                         ELSE TEST:=5;
     RPMBIT:=FALSE;
      CASE TEST OF
          1:BEGIN {***TRT <=14.4***}
              HT 1300:=0.0;
              HT 2400:=0.0
            END;
          2:BEGIN {***14.4 <TRT <=23.4***}
              HT 1300:=TRT-14.4;
              HT 2400:=0.0
             END;
          3:BEGIN {***23.4 <TRT <=28.4***}
              HT 1300:=9.0;
              HT 2400:=TRT-23.4
             END;
          4:BEGIN {***28.4 <TRT <=32.4***}
              RPMBIT:=TRVE;
              HT 1300:=TRT-20.4;
              HT 2400:=0.0
             END; {***32.4 <TRT***}
          5:BEGIN
              RPMBIT:=TRUE;
              HT 1300:=0.33 *TRT;
              HT 2400:=0.67 *TRT-20.4
             END
          END{CASE}
          .
          .
          .
```

Figure 1b

VARIABLE TYPES IN PASCAL

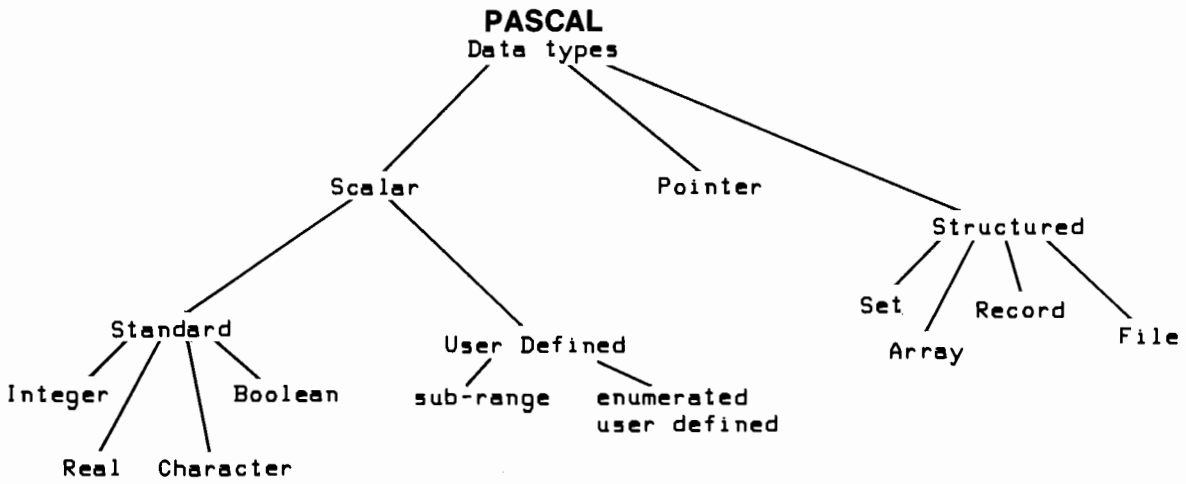
PASCAL variables can be associated with a data type, i.e., a set of data objects, such as whole numbers, that are alike in some way. PASCAL offers a very extensive set of data types. They are shown in figure 2.

The scalar types can be the pre-declared types of Integer, Real, Characters and Boolean, plus the user-defined scalars known as the "subrange" type and "enumerated user defined" type. The structured data types can be sets, arrays, records and files. In addition there is a dynamic data type defined as a "pointer".

The existence of these types contributes to the power of PASCAL in two ways. First, a problem can be expressed more clearly and precisely if appropriate type definitions are employed, and this makes reading and writing programs easier. Secondly, type declaration gives more information to the compiler about the problem, and the compiler can make use of this information to make more extensive error checks, and produce more efficient programs.

THE CONCENT OF DATA TYPES

A DATA TYPE is a set of data objects, such as whole numbers, that are alike in some way.



Scalar or unstructured - defines or enumerates all the possible constants of that type. The enumeration is ordered

Structured - formed by combining scalar data types.

Figure 2.

PASCAL DATA STRUCTURES

PASCAL data structures can be divided into unstructured data types (scalers), structured data types: sets, arrays, records and files, and a dynamic data type, the pointer. PASCAL data structures allow us to express algorithms more naturally, and offer greater compile time security than does any other sequented programming language in use.

Scalar Types As mentioned before, the standard scalar types are integer, real, boolean and character. PASCAL allows the user to define additional scalar types. This is done by inventing a name for the type, and names for all of the values one wished to associate with that type.

```
TYPE
  chess_pieces=(pawn, knight, bishop, rook, queen, king);
  sizes=(small, medium,  large);
  search_states=(key_assent, key_found, searching);
  flow_points=(F101, F102, F103, F104, F105);
  day=(monday, tuesday, wednesday, thursday, friday, saturday);
```

Using these type definitions, we could declare the following variables in our program:

```
VAR
  holiday, workday: day;
  temp: flow_points;
```

It is also possible to define a subrange type that provides a lower bound and an upper bound in the associated integer, character, or scaler types.

```
TYPE
  letter='a'..'z';
  digit='0'..'9';
  weekday=monday..friday;
```

Subrange types provide for an important testing facility in the PASCAL language. Range checking is an important, and often neglected, aspect of computer programming. Subranges take the burden of range checking away from the programmer, and pass it on to the compiler. The use of subrange declarations also improves the readability of the program. By telling the reader the range of values a variable can take, you tell him something about that variable's use.

Structured Data Types A structured type differs from a simple type, in that the variables of a structured type have more than one component. Each component of a structured type is a variable which likewise may have a simple or structured type. At the lowest level, the components of a structured variable have simple types, and these may be assigned values and used in expressions, in the same way as simple variables.

There are four structured data types in PASCAL: sets, arrays, records and files.

The record is a very powerful data structure. The record and the array are structured variables with several components. The components of a record may have different types, meanwhile, the components of an array are all of the same type. The components of a record are accessed by name instead of by subscript, as in the array. Record types are often used to describe tree like data structures, where each node of the tree is named by a distinct selector name. An example of such a structure is shown below.

```
TYPE date=record
                day:1..31;
                month:1..12;
                year:1000..2000
          end;
```

This code declares that a record type DATE is a three-component structure, whose individual components may be referred by the selector names; day, month, and year. For example:

```
VAR
BEGIN
  x.day:=7;
  x.month:=5;
  x.year:=1976;
```

An array is an ordered collection of variables, all of which have the same type. A line of text can be represented as an array of characters; a vector can be represented as an array of real numbers, and since a matrix consists of columns, each of which is a vector, a matrix can be represented as an array of vectors.

An array type is declared in terms of an index type and a base type.

```
TYPE
     direction=(x,y,z);
     vector=ARRAY[direction] of REAL;

VAR
     s,t:direction;
     w,v:vector;
     buffer:ARRAY[1..maxler] OF char;
```

An array with base type, "boolean" has the same properties as a set. Each component of the array corresponds to a potential member of the set, which may be absent (false), or present (true).

```
TYPE
  CONTACTS=
     (CONTACT_ONE, CONTACT_TWO, CONTACT_THREE....,CONTACT_N);
VAR
     CONT:CONTACTS;
CONT_SET:SET OF CONTACTS;
CONT_ARR:ARRAY[CONT]OF BOOLEAN;
```

Sets of variables may be declared. If S is a set of objects of type T, then any object of type T is either a member of the set, or not a member of the set. Examples:

```
TYPE
  integer_set=set of 0..30;

  attributes=(corner_lot, near_car_line,
              close_in, shade_trees,
              nearby_shops);

  property_description=set of attributes;
  contact=(C100,C101,C102,C104);
  contact_set=set of contacts;
  letters=('a'..'z');
  digits=('0'..'9');
  letter_set=set of letters;
  digits_set=set of digits;
```

Some examples of using sets are shown below:

1. A possible statement to determine if a character is a member of a set of characters, could be:

```
  read (ch)
  IF (ch='E') OR (ch='C') OR (ch='S')
OR (ch='I') THEN...
can be expressed as
  IF ch IN ['C','E','I','S'] then...
```

2. The expression to determine if a character is a digit or not, would normally be

```
IF (ch>'0') AND (ch<='9')...
```

This can be coded as

```
IF ch IN ['0'..'9']...
```

As has been shown here, PASCAL is a well designed language for structured programming. As a result, this powerful language is easy to write, easy to read, and perhaps most importantly, easy to maintain.

BIBLIOGRAPHY

Grogono, Peter - Programming in PASCAL, Addison-Wesley, 1978

Webster, C.A.G. - Introduction to PASCAL, Heyden, 1976

Wilson, I.R., Addyman, A.M. - A Practical Introduction to PASCAL, Springer Verlay, 1979

Wirth, Niklaus - Algorithms + Data Structures = Programs, Prentice - Hall, 1976

Kieburtz, R.B. - Structured Programming and Problem Solving with PASCAL, Prentice - Hall, 1978

Jensen, K. - Wirth, N. - PASCAL - User Manual and Reprint, Springer - Verlay, 1974

Conway, R., Gries, D., Simmerman, E.C. - A Primer on PASCAL, Winthrop, 1976

Schneider, Weingart, Perlman - An Introduction to Programming and Problem Solving with PASCAL, Wiley, 1978

HP Subroutine Linkage Conventions

Robert Niland/HP Lexington

[Editor's Note: This is part two of a six part series. The introduction and a description of the JSB instruction appeared in Volume III, Issue 6 of the Communicator.]

3-1. SUBROUTINE RETURN

Upon completion of the functional code, the final task of a subroutine is to return to the caller. In a subroutine which was entered via a JSB instruction, the address of this return point is in the subroutine's entry point. The subroutine need only transfer control through its own entry point, and program execution will resume at the next instruction following the caller's original JSB. A typical entry/exit sequence is:

```
SUBRU MPX             Entry point/return address.
      ...
      ...             Task-specific code.
      ...
EXIT  JMP SUBRU,I     Return to caller. ("EXIT" is optional)
```

The "transfer through" is accomplished with the JMP (JuMP) indirect instruction. The JMP instruction has the same bit structure as the JSB with the exception of the opcode bits (13-11) which are 0101 binary. There are several other differences between the JSB and JMP instructions. When executed, the JMP resets P register to the final operand address (not address +1). It does not modify memory contents at the destination location. The indirect bit is set to indicate that control is not to be passed to location SUBRU, but through it. If locations SUBRU and EXIT do not reside on the same page, the JMP operand address will actually point to a link, and the link's contents will be the address of location SUBRU. However, since the user specified indirect, both the JMP and the link will have bit 15 set. Note that in RTE systems, the relocating LOADR usually creates such links on page 0, the base-page. As described in Section II, the return address written into the entry point (SUBRU) by the caller's JSB, will remain unmodified until entry from the next calling location in the program, or until the location is explicitly written to by other instructions. This means that our "trick" of employing the MPX (halt) in the entry point to trap illegal entry will only work if the defective call is the first call to the subroutine. One way to obtain the MPX protection on every call (perhaps while testing), is to write an MPX into the entry point before returning. The following is one of several ways to accomplish this.

```
SUBRU MPX             Entry point/return address.
      ...
      ...             Body of subroutine.
      ...
      LDA SUBRU       Get return address from entry point.
      ...
      LDB .MPX        Get a new "MPX" halt instruction.
      STB SUBRU       Reset entry point to MPX.
      ...
EXIT  JMP A,I         Return through A register.
      ...             Subroutine variables & constants follow.
.MPX  MPX             Constant 102000B
A     EQU 000000B     A register is memory location 0.
```

3-2. REGISTER PARAMETER PASSING

It is a rare subroutine indeed which requires no information from its caller, and passes back none upon returning. In almost all cases information must flow in one or both directions. The simplest form of parameter or argument passing on the HP1000 takes advantage of the fact that neither the JSB nor the JMP instructions disturb the user's working registers, which are:

A,B: 16 bit accumulator and I/O registers. Can be linked for double word/bit manipulations and are addressable as memory locations 000000B and 000001B.

E,O: 1 bit Extend and Overflow registers. Not typically used to pass data, although a subroutine might well be written to test or set them.

S: 16 bit switch register (I/O location 01B). Not recommended for parameter passing, since ALL programs on the system share this register and are not protected against one another's use of it.

X,Y: 16 bit index registers. Accessible via index instructions only on HP1000-M, E and F series CPU's. Note: early revisions of RTE-II do not save and restore these registers during interrupts.

The only requirements for passing register parameters to a subroutine are:

1. The register(s) must be loaded at some time prior to the JSB or JMP.

2. They must not be disturbed between step 1. and the JSB/JMP.

With respect to requirement 2, it should be noted that preservation or restoration of registers A,B,E,O,X and Y is guaranteed whenever the program is interrupted by events in the system which are not caused by instructions the program is currently executing. However, when a program makes a call to any HP system, subsystem, or library entry point, register preservation/ restoration is only guaranteed to the extent documented for that particular call, with the sole exception of the S register, which RTE never uses. For example, calls to EXEC (i.e. JSB EXEC) are documented for A & B, but not E, O, X & Y. The user must assume that the contents of E...Y will be lost.

3-3. REGISTER PASSING EXAMPLE

This section will discuss a typical subroutine which uses simple register parameter passing. This subroutine was written to split the 16 bit contents of the A register into two 8 bit bytes and return them separately. A fully documented source listing of .SBA appears in Appendix B. The calling sequence for this routine is that the A register contain the object 16-bit word, and that the B register is scratch (i.e. has been saved if valuable).

```
...             Somewhere in calling routine.
LDA bytes       A has been loaded with byte pair.
...             Other processing not affecting A.
JSB .SBA        Invoke routine to Split Bytes in A.
...             Return point.
STA right       Save right 8 bits (for example).
...
STB left        Save left 8 bits (for example).
...             Program continues & exits.
...
```

The subroutine to perform this operation is:

```
.SBA    MPX             Entry point/return address, with [A].
        CLB             Clear all of B to zeros.
        LSL 08          Move A-left into B-right, A-right to A-left,
*                       clearing A-right to zeroes.
        ALF,ALF         Move A-left back into A-right.
EXIT    JMP .SBA,I      Return with bytes in A & B.
```

3-4. INSTRUCTION EMULATION

The use of the machine registers to pass information to subroutines, seen in the preceding sections, limits the amount of data to whatever can be passed in the four 16-bit registers. Additionally, the use of those registers forces the calling program to save their previous contents prior to the JSB. One technique which, although still at the Assembly level, can extend the options available to the programmer, is to take advantage of a memory area which is known to both the calling routine and the subroutine for parameter storage. Since the subordinate routine 'knows' the return point in the caller, it is fairly simple to reserve memory locations in the vicinity of the JSB which are to be used for parameters. The reserved locations are usually the one or more words which immediately follow the JSB. This method is used extensively by HP microcode for the double and triple word instructions, and by various library and instruction emulation routines. For example, the DLD (Double LoaD) instruction standard on current HP1000 computers was an option on some earlier 21xx series machines (such as the 2116A). If it were necessary to write software compatible with this older CPU, the DLD instruction could not be used. It would have to be emulated instead.

Suppose the sequence.....

```
ASMB,F....
       ...
NULLS OCT 000000,000000
       ...
       DLD NULLS
       ...
```

is desired, and the DLD is at location 34067B, and NULLS is symbolic for location 34021B. In memory, the HP1000 instruction would look like this:

```
34020 ...
34021 000000        (NULLS) 1st of
34022 000000        two words.
34023 ...
       ...
34066 ...
34067 104200        DLD opcode.
34070 034021        Operand address (WORDS)
34071 ...
```

The DLD will be emulated by replacing the ASMB opcode 'DLD' with the following sequence:

```
ASMB,....
       ...
       EXT .DLD
       ...
NULLS  OCT 000000,000000
       ...
       JSB .DLD       Replace opcode with subroutine call.
       DEF NULLS      Have operand address assembled here.
       ...
```

In memory this will look like..

```
34020 ...
34021 000000         Same
34022 000000         Same
34023 ...
       ...
34066 ...
34067 014xxx         The JSB (to some EXT location).
34070 034021         Address of NULLS (just as in DLD).
34071 ...
```

Our subroutine (.DLD) might look like this..

```
ASMB,L,T,C
       NAM .DLD,7 790608 Emulate DLD for the 2116A
       ENT .DLD
B      EQU 000001B          The [B] register.
       MIC MPX,102000B,0    Trap any JMP .DLD
*
.DLD   MPX                  Entry point-raw return address.
*
       LDB .DLD             Get address of DEF NULLS.
.NEXT  LDB .DLD,I           Get link-to/address-of NULLS.
       RBL,CLE,SLB,ERB      Skip if a direct address!
       JMP .NEXT            Otherwise fetch next level of indirect.
*
       LDA B,I              Get first word of pair.
       INB                  Point at next.
       LDB B,I              Get second.
*
       ISZ .DLD             Increment return to skip DEF NULLS.
       JMP .DLD,I           Return to DEF NULLS + 1 (34071).
       END
```

The above subroutine should be taken as an example of the emulation concept only. It does not precisely emulate the DLD instruction for two reasons:

1. The E register is not preserved.

2. If the operand were DEF B,I and the [B] register contained the final address 34021B, the routine would fail because the contents of [B] are destroyed at step .NEXT.

Finally, the preceding is not the source code for the .DLD routine in the DOS/RTE Relocatable Library. That routine (if called) assumes that the caller is executing in a machine which has a DLD instruction. It replaces the JSB .DLD with an octal 104200 and executes it. On subsequent executions of the same location, the machine's DLD is executed directly. The replacement is done because both DOS and RTE require that the computer have the instruction set of which DLD is a member. This replacement is accomplished by replacing the emulation routine with a routine of the same name which does not emulate, but instead calls library routine .MAC. In our example the working code of .DLD would be replaced by

```
ASMB,L,T,C
       NAM .DLD,7  790610 Install DLD machine code.
      MIC MPX,102000B,0  Not changed.
.DLD  MPX                Entry point.
      JSB .MAC.          Replace caller's JSB .DLD
      OCT 102400         with DLD machine instruction.
      END                Return is from .MAC.
```

Although this method of replacing emulation calls is effective when required, it is not as efficient as other techniques which are available, and which are discussed in section 3-5. The concept of run-time replacement deserves understanding, because there are situations in which it is the only way to link a software call to the installed firmware. An example of such a case is presented in section 3-6. In the dicussion of instruction emulation, we saw an example (.DLD) which uses adjacent memory to contain the address of the operand. The use of these locations is not restricted to addresses. For example, repetitive instructions which might take longer than 75 microseconds to complete, must periodically check the Interrupt Register, and if an interrupt is logged, must save their state and allow the interrupt to be serviced. Since only the working user registers (A,B,E,O,X & Y) are saved by the operating system, any additional information about the state of the interrupted instruction must be saved in memory. The HP1000 MoVe-Words (MVW) is such an instruction. Were we to emulate it in software, the calling sequence would have to be:

```
LDA FROM        Get (direct) address of "from" location.
...
LDB TO          Get (direct) address of "to" location.
...
JSB .MVW         Call routine.
DEF WORDS [,I]   Define address of move count.
NOP              Reserve memory location for micro-code.
...              Return point.
```

Refer to the listing of routine .SFW in the appendix for an example of an emulation subroutine.

3-5. CODE REPLACEMENT

When software which contains JSB <emulation routine> calls is to be executed on a computer which has a machine instruction for that operation, there are three ways to invoke the use of the hardware/firmware:

1. Generation: RTxGN 'RP' command during 'CHANGE ENTS' phase. RELocate a code-replacement module containing ASMB 'RPL' pseudo-instructions during 'PROGRAM INPUT' phase.

2. Relocation: Use LOADR commands RE, SE, Or LI to relocate 'RPL' modules.

3. Execution: Emulation routine can call .MAC. as discussed in sections 3-4 and 3-6.

Methods 1 and 2 are very similar. Both result in the creation of RPL relocatable records (IDENT = 010 binary, RELOCATION INDICATOR = 4). The only difference is that the RTxGN 'RP' command, described in your Generator manual for the "CHANGE ENTS" phase, also purges from the final system any module having an ENT record which matches the mnemonic in the command. When a program is loaded, either by RTxGN or LOADR, RPL modules referenced as part of a library, (LI,), relocated (RE,) or searched (SE,SEA,) are not actually 'relocated' or made part of the program's memory image. The RPL module is informational only. It tells the loader that any instruction (typically a JSB) referencing an external (EXT) symbol which is satisfied by (found in) an RPL module, is NOT to have its (the instruction's) address field set to a memory location, but instead, the referencing instruction is to be REPLACED by the RPL's associated value. This means that a...

```
JSB .DLD
```

source instruction which assembled as...

```
014000X  JSB .DLD
```

and which might load as...

```
016371   If .DLD were at 371B current page
```

in a system not having a DLD instruction, would load as...

```
102400   (a DLD machine instruction!)
```

in a system generated with .DLD,RP,102400 or in a system where the following module was assembled and relocated with the program containing the JSB .DLD

```
ASMB,L,T,C
       NAM .DLD,7   790610 change JSB .DLD to 102400B
       ENT .DLD            To satisfy the caller's EXT.
.DLD   RPL 102400B         For RTxGN's & LOADR's information.
       END
```

RPL modules can contain any number of ENT's, and multiple modules can be combined for use in generations or during on-line loads. They need NOT be SEarched by LOADR. Because they consume no program address space, it is perfectly safe to RELocate such RPL 'libraries' with any and all programs, whether the programs require them or not. However, it will result in a long list of entry points if the FM,LE (ForMat, List Entries) command (or its antecede equivalent) is used. The RPL capability of RTE is normally used to "tell" the RTxGN generator and the LOADR what instructions are available on the computer. However, since some versions of RTE require certain instructions to be present (e.g. mapped RTE's must have the D.M.I. group), the RPL can be used to simplify application code which is operating system dependent. For example, suppose you want to write a random number generating subroutine which uses the value of the least significant word of the system clock as the initial seed. This value is found in system entry point $TIME. In unmapped RTE's (e.g. RTE-B,C,I,II,M1,M2), this value can be obtained by a simple

```
LDA $TIME
```

In mapped RTE's (e.g. RTE-III,M3,IV,IVB), $TIME is in the system map. It may also be in the user map, but is safest to obtain by using

```
XLA $TIME
```

How does our subroutine determine which instruction to use? Well, we could include both, and select the proper one based on a a call the to library routine .OPSY, so let's make our subroutine do this. But let's call the subroutine .XLA and make its calling sequence

```
...
JSB .XLA        Call the routine.
DEF $TIME       (or more generally DEF <symbl> [,I])
...             Return point.
```

When in an unmapped RTE, we will use the subroutine, but in a mapped RTE, we can either use it, or eliminate it entirely by including a

```
.XLA,RP,101724        *X-Load the A register.
```

during the system generation. In this manner, our random number utility can be written for any RTE, by accessing $TIME with a JSB to .XLA. The software .XLA will always work, but in a mapped RTE the call can take less space and run faster because the software .XLA may be replaced by a real XLA instruction. Note that the software .XLA in this example only simulates the XLA in an unmapped environment. In a mapped environment it either emulates it, or is replaced by the emulated instruction. Since unmapped RTE's would never require true emulation of a mapping instruction, it does not matter that only an LDA is actually being performed. Needless to say, you must not include a .XLA,RP,101724 in your unmapped RTE generations even if the computer has Dynamic Mapping installed. On the other hand, you may as well RP the entire installed set in your mapped RTE generations.

3-6. RUN-TIME REPLACEMENT

In a computer product family like the HP1000, where software has to be upward compatible with, yet be able to take advantage of, the growing selection of firmware enhancements, a means has to exist which links the software to whatever capability the computer has. We have seen an early solution in the discussion about .MAC. in section 3-4. It was not entirely satisfactory for two reasons:

1. The code replacement was done during execution, slowing the program, and consuming space.

2. It always assumed that the firmware existed.

The current method utilizing RPL relocatable records, discussed in section 3-5 does not have these problems, but it does have two of its own:

1. The system manager/programmer must "tell" either the Generator or the LOADR what firmware instructions are installed. This is a minor disadvantage that could be eliminated by coding the LOADR to test the computer and programmatically determine what is present, at the expense of LOADR size & speed. It would not be wise for the Generator to do this, as the system might be intended for use on different hardware.

2. The RP/RPL method can only be used for references which are replaced by a single 16 bit firmware op-code.

Objection 2 has become significant with the advent of the Vector Instruction Set, in which about half the calls have a two-word opcode. Since RTE has no facility to produce or process a 2-word RPL record type, the code replacement has to be done at run-time. It is being done by two routines in ⅔VLIB, the vector library. They are .VSRP and .VDRP. In order to avoid the drawbacks of .MAC., these routines have two features:

1. Multiple entry points: .VSRP handles replacement for VADD, VSUB, VMPY, VDIV, VSAD, VSSB, VSMY and VSDV, eliminating the need for a separate replacement routine for each call. Based on which entry point the user's JSB comes in through, .VSRP selects the appropriate sub-opcode, and stores it in the location just after the user's JSB.

2. The first words of the opcode pair for each of these calls are the same (101460B), so .VSRP, in order to avoid assumtions about its hardware environment, uses the existing RPL facility of RTE to determine what is installed. It has a DEF .VECT+0 in its code. If .VECT has been RP'd or RPL'd to 101460B during generation or load, .VSRP overlays the user's JSB with the 101460B. If instead, a software version of .VECT has been relocated, .VSRP stores the user's return address in its entry point, and jumps to .VECT+1, simulating a JSB directly from the user. .VECT must examine the sub-opcode stored in the caller to determine which V.I.S. routine was called. See the following note.

Heavy users of micro-programming, who find themselves running out of opcodes, and who must therefore resort to double-word opcodes, should consider this method.

.VDRP is identical to .VSRP except for entry point names.

NOTE

Concerning software simulation of V.I.S instructions. There is at present no software version of .VECT. If a user who does not have V.I.S. firmware wishes to write code compatible with V.I.S. it is recommended that the Fortran rough-equivalents listed in the V.I.S. manual (12824-90001) be used. An experienced programmer could write more efficient versions of these routines, and write a software .VECT interface routine, but be advised that the V.I.S. micro-code uses memory pre-fetch or "pipelining". This means that any software which precisely emulates the firmware will likely be slower than the Fortran code the firmware was designed to replace, and any software which only simulates the firmware (such as the manual examples) will perform differently in some circumstances, particularly if the destination array is also one of, or part of one of, the source arrays.

A BRIEF INTRODUCTION TO THE C PROGRAMMING LANGUAGE

Tim Chase/Corporate Computer Systems, Inc.

While in graduate school I took a course in compilers. In that course, the professor commented that there were just too many different programming languages. What the computing world needed, he said, was some consolidation. To do this, he proposed his own new language....

It is in this light that we will describe what to many of you, will be yet another new language. The name of the language is C. We will present C as unbiasedly as any manufacturer can present what he believes to be a fantastic product. The natural inclination will be for readers to compare C with the language they currently love. We think that C can hold up under close scrutiny, for it is a lean and powerful language with a great deal of flexibility.

WHAT C IS

C is a general purpose language developed by Dennis Ritchie at Bell Telephone Laboratories. Its history may be traced from BCPL through several home-brew languages. Its immediate predecessor was named, logically enough, B, which has evolved to the present name, C.

As a language, C is unique because the objects it deals with, and the operations it performs, map easily into those found in modern computers. C allows the programmer the usual integers and floating point numbers, but it also allows access to characters and even machine addresses. This makes C a good choice for "systems programming", especially in the RTE.

C supports all the modern control structures which make structured programming easy. There are only 15 different statements so the entire instruction set of C can be quickly mastered, and in addition, the flexibility of each of the statements allows C to address complex problems.

The HP/C compiler has been designed to interface to the existing HP software base, and it does so in ways better than any currently available System/1000 language.

WHAT C IS NOT

C is not a paternalistic language like PL/1 which takes care of the programmer no matter what, and at the same time restricts what can be done. Instead, C is more in philosophy like an assembler. The programmer can say almost anything in the language, and the resulting code is quick and efficient.

C does not have any intrinsic input/output statements. This is good and at the same time bad. I/O is so much a part of the philosophy of the host operating system, C's designers felt that it should not be a part of the language. However, C's run time library contains many I/O routines which do formatted input and output as well as binary I/O. These routines, as might be expected, far exceed the formatting capabilities normally found in FORTRAN/BASIC type languages.

A BRIEF OVERVIEW

C is vaguely reminiscent of ALGOL or PASCAL. It is a free formatted, block structured language with well typed variables. The objects (variables) in the language behave with a powerful ALGOL-like scope, so that objects with the same name may be declared within nested blocks without confusion.

Consider the following example:

```
HPC,1,m,"DEMO1,3 FIRST DEMONSTRATION";

/* This is our first C program */

main()
{
  static int x[] = { 1,2,3,4,5,6,7 };
  static int i;

  s=0;
  for( i=0; i<sizeof(x)/2; i=i+1)
    s = s + x[i];
  printf("The sum of x is %d \n", s);
}
```

What do we have? The first line is the compiler control line. It specifies some of the options and features that apply to the module being compiled. C has about 10 options to allow versatile control of the compiler. We have requested a listing (l) and a mixed listing (m). In addition, we have named the module DEMO1 with the relocatable NAM record comment FIRST DEMONSTRATION.

Comments in C, as in some other languages, may appear anywhere between language tokens by enclosing them within matching /* and */ pairs.

Every program must have a "main". This is much like the FORTRAN "program" statement. The "()" after the main indicates that it is a program. The characters "()" appear for every function, even if there are no arguments. They may be thought of as performing the operation of "calling" the function or subprogram.

The { is the beginning of a block. In this case it is the block which holds the definition of the main program. The { is the same as BEGIN in other languages and is easier to type. Likewise the } is used to close a block much like the END statement in ALGOL.

The first two statements of the example are declaratives. They specify the "storage class" and the "type" of the objects. The storage class describes how the object is to be stored in memory. Static tells the compiler that the object is to be assigned to a given storage location permanently. If we had specified "auto", then the object would have been assigned space on the run time stack, and would have been dynamically created and destroyed as necessary.

Note that we have declared x to be an array by using the "[]" characters after it. C has a rich grammar for defining objects, which allows the programmer to describe arbitrarily complex objects — in fact the programmer can even invent (within bounds) his own objects.

The type of the variable x is integer (as specified by the "int") and the dimension for x is found by counting the initial values assigned to x. In the case of this example, x has 7 locations with the integers 1 through 7 in them. If another value were entered between the {}, then the compiler would increase the size of x to hold it.

Had we wished to just reserve 7 locations without putting something into them we could have written:

```
static int x[7];
```

We also could have written something like this:

```
static int x[SIZE];
```

and then defined SIZE elsewhere. In fact, we can even use constant expressions for dimensions:

```
static int x[SOME+MORE/2];
```

Continuing with the example, we set the value of s equal to 0. Note that s has not been declared so C defaults it to an integer. In addition, if we had so requested, C would have issued a warning message indicating that it was defaulting s. This feature helps indicate where spelling errors may have been made.

The "for" statement syntax is unique to C. The statement has 3 parts. The first is the initialization. We have set the location "i" to 0. Next is the test part. The test is done before each iteration of the loop. Here we check to see that the value of i is less than the dimension of x. The "sizeof" function is actually a constant equal to the number of bytes in the specified object. We divided by 2 to get the number of words. The operations "sizeof" and division by 2 are done at compile time so the resulting expression is really "i<n" where n is the size of x in words. The third and last part of the "for" statement is the increment part. After the loop is executed an increment is done on i, and the loop test is retried. We have elected to increment i by 1 in traditional DO loop fashion.

The real power of the "for" statement is in that the initialization, test, and increment expressions are general. For example, we can make a "for" loop which walks a linked list by making the initialization set a pointer to the first element of the list, make the test to see if the pointer is null (end of list), and make the increment set the pointer to the address of the next element of the list.

In our example, the only statement which composes the loop adds x[i] to s.

The function "printf" is for output. It passes a format string (enclosed in quotes) to the formatter, along with a list of variables (in this case only s). The format specification for converting s to decimal is the "%d" pair of characters. The "\n" at the end of the string tells the C I/O system to output an end of record.

Finally, falling off the end of the main causes the program to terminate.

Increasing in complexity, consider the following example:

```
HPC,L,"DUMP Dump Memory";

main()
{
  int *addr, count;

  printf("Start address and count\n");
  scanf("%o %d",&addr, &count);
  do {
    printf(" %06o %06o \n", addr, *addr);
    ++ addr;
  }
  while ( -- count);
}
```

Now we are getting somewhere. This program is an absolute memory dump which will print a block of memory on the user's terminal. It makes use of pointers and a new I/O function called "scanf".

The declaration

```
int *addr, count;
```

declares the objects "addr" and "count". "Count" is an integer, but the "*" character causes the object "addr" to be declared as a pointer to an integer. This means that the 16 bit contents of "addr" contains the memory location address of an integer. As you might expect, we can have pointers to integers, floats, characters and even pointers to functions.

The "printf" function requests values for the "addr" variable and the integer "count". Calling "scanf" reads the data into the "addr" and the "count" variables according to the format string. The "%o" specifies octal while the "%d" specifies decimal.

A "do" statement follows the "scanf" call. "Do" is a "bottom of loop check" statement. This means that the statement following the "do" will be executed once, regardless of the outcome of the test in the "do" statement. In other words, the test is checked at the bottom of the loop. The statement in the do is a compound statement. In C, a group of statements may appear anywhere a single statement may appear, by enclosing the group in "{}" characters. In this case we have the "printf" call and the "++addr" statement grouped together as one.

The "printf" call prints out the variable "addr" and also the contents of the memory location pointed to by addr. The unary "*" operator signifies indirection (it is comparable to the ",I" in assembler). The format used in the "printf" call is a bit different than before. The "%06o" signifies that the octal field is to be at least 6 positions wide, and should be zero filled. If we had written "%6o", the field would have been 6 positions wide, but blank filled. If we had written "%o", it would have been the number of positions necessary to hold the number being printed.

The statement following "printf" is unique to C, and very useful. The "++" operation (as used here) is an instruction to increment an object by one. Writing "++x" is about the same as writing "x=x+1", but the former generates more efficient code. The "++" is a powerful construct, since it may be used in several different expressions. For example, we could write

```
y = ++x + z;
```

This says "take x and increment it, store the incremented value back in x, then add z to that value and store the result in y." This capability is especially useful in loops.

If we write the "++" after the object, it takes on a different meaning, but one that would naturally be expected. The expression:

```
y = x++ + z;
```

says "Take x and save it, increment the memory copy of x, add the saved (unincremented) version of x to z and store the sum in y." The "++" after the x is called a "postfix increment", while the "++" in front of x is a "prefix increment".

At the end of the "do" statement is the "while(--count)" clause. The "--" operation is like the "++" except that it decrements the argument. The net result of the "do" statement shown in the example is that the "printf" is executed and the "count" variable is decremented. If the resulting "count" value is non-zero, then the "printf" is executed again. Note that by using the "--" we are able to decrement the "count" variable and test it in the same expression.

RTE AND C

C's power becomes surprising when it is used to interface with existing data structures. The RTE operating system is loaded with information filled tables and structures which the skillful programmer can use to his advantage. This is in itself a problem because of occasional revisions in the RTE. Tables come and go and sometimes change in layout. C allows the user to describe many of the important features of the tables independently of the code which uses them. As an example, consider the following:

```
HPC,1,"EXAMP Structure Example";

struct {
  char file[6];        /* File name */
  int  type;           /* File type */
  int  track;          /* Starting track */
  int  extnt_no:8;     /* Extent number */
  int  st_sec:8;       /* Starting sector */
  int  num_secst;      /* Number of sectors */
  int  rec_leng;       /* Record length */
  int  s_code;         /* Security code */
  int  open_flags[7]; /* Room for open flags */
       } dir;

main()
{
   :
 exec(READ,lu, dir, -sizeof(dir) );
 if( dir.type == 4) {..... }
   :
```

The code above is not a complete program, but rather a fragment designed to show how easily C interfaces to the RTE as it exists today, and allows for changes in the RTE tomorrow.

The "struct" keyword tells C that we are going to define a "new type". The storage class keywords may be applied as appropriate to the new type. In this case we have used the default storage class.

The "struct" statement has several forms. We have used the most simple form in the program above. The format is:

```
struct { <sdefs> } <list>
```

The "<sdefs>" define the members of the structure, while the "<list>" defines the variables which will have these members. We chose to describe the FMGR disc directory entry with the "<sdefs>". The only object we defined was a single location called "dir". "Dir" is special, however, since it is really composed of all of the objects listed in the "<sdefs>" used to define it. This means that "dir" really has 16 computer words associated with it. This structure can be considered an implementation of the concept of extended arithmetic, since there are several words associated with each extended variable. There is a problem with "dir", however. We must be able to specify which member we are currently interested in. This is done in C by using the "member of" operator represented by the "." character. If, for example, we are interested in the type associated with the directory entry currently in "dir", we would write:

```
dir.type
```

This "qualified name" may be used anywhere a normal name could be used. The example program fragment above shows this. We read something into "dir" (note that the length of "dir" in bytes may be gotten from the "sizeof" operation. This capability allows changes in the definition of "dir" without recoding. The "if" statement following the EXEC read checks to see if the type of the entry in "dir" is equal to 4. If the comparison is true, then some action can be taken.

By using structures, we may describe the FMGR directory construct in a way which may be easily changed. If the location of the file type were to change, we would only have to change the definition of the "dir" structure, and the program would still work since file type specifications in the directory are referenced by the name "type," and not some absolute offset into the "dir" object.

This same technique is very useful with IMAGE. Structures may be used to define the field layout of data records within an IMAGE data set. When the data set is changed, the structures may be changed, yet the programs will all still work correctly. This is a far cry from the use of EQUIVALENCE statements in FORTRAN.

We declared "dir" to be a scaler structure. We can also declare arrays of structures. If "dir" had been written "dir[8]", that would have defined an array of 8 directory entries. The dot operator would have worked in the same way, but we would have had to specify which of the 8 directory entries we were interested in. For example, to reference the starting track of the fourth directory entry, we would write:

```
dir[3].track
```

Remember that arrays all start at 0 in C.

Notice also the specification for the file name, it is an array. If we had an array of 8 directory entries (this corresponds to one FMGR block) and we wanted to set the third character of the fourth directory entry to an 'X', we would write:

```
dir[3].file[2] = 'X';
```

Or if we wanted to set the file names for each of the 8 directory entries to all blanks, we could write:

```
for(i=0; i<8; i++)
 for(j=0; j<6; j++)
   dir[i].file[j] = ' ';
```

As a final note, before moving on to other things, notice the definition of the structure members "extnt_no" and "st_sec". These are called "bit fields". Bit fields are strips of bits which are shorter than a word in length. They are packed left to right in 16 bit words and contain unsigned integers. In our example declaration, we have made "extnt_no", 8 bits in length, and occupy the left most position in the word. "St_sec" is also 8 bits in length, and it is in the right half of the word.

Bit fields are especially useful for interfacing C programs to existing data structures in the RTE. Their use allows for easy alterations later. For example, assume that in RTE-IMCMXLLV the FMGR will allow 32,000 extents instead of only 256, as it does now. The above program, written using C, will easily accomodate the change since it only needs to have the "dir" definition changed to make "extnt_no" an integer instead of a bit field (effectively remove the ":8" designation).

MACROS IN C

One of the most useful and unique features of C is that it has a built in macro capability. Macros are typically found in assemblers. In higher level languages they are rare, but just as useful.

Consider the following example using macros:

```
HPC,p,1,"MACRO Macro Example";

#define SIZE 100
#define READ(lu,buf) exec(1,lu,*buf,-sizeof(buf))

main()
{
  int line[SIZE];
   :
  READ(1, line);
   :
```

The above program fragment shows some simple uses of the macro pre-processor. Essentially, the macro pre-processor is an optional first pass over the C source, which does only text processing (and not any actual code generation) to produce a new source file which can then be compiled by HP/C.

The example we have shown defines 2 macros. One is a simple macro called SIZE, while READ is a more complex macro which uses macro parameter substitution.

By using the macro pre-pass, we can allow the variable name SIZE to represent the integer 100. This provides a method of abstracting constants (and even character strings) so that they may be changed without changing the whole program. If we wanted to change the dimension for "line" above, we would only have to change the definition of SIZE.

Unlike other languages, this is not a "constant" definition. We may substitute any characters for the macro name. We could, for example, define a new statement:

```
define repeat while(1)
```

This would make the macro "repeat" become while(1) to allow us to code an infinite loop:

```
repeat {
   a += x[i++];
   if(i == 100) goto out;
}
```

Simple macros make programs more readable since they allow the C language to be extended to accomodate what is being represented in a more natural form.

READ in the above example is a macro which uses macro parameter substitution. READ has been defined as having 2 parameters, "lu" and "buf". Following the list of parameters is READ's definition. READ is defined to be a call to EXEC. In the example, the line "READ(1,line)" will be expanded to be:

```
exec(1, 1, *line, -sizeof(line))
```

Where "lu" was used in READ's definition, 1 will be substituted and where "buf" was used in READ's definition, "line" will be substituted.

Macros allow simplification, yet still generate efficient object code since they are "in-line constructs", and don't call subroutines. Through their use, C can be transformed into a special purpose language for use even by novice programmers. The "include" feature of the macro pre-processor makes the use of macros especially attractive.

"Include" allows you to have portions of C programs in different files, and then bring the files together at compile time. For example, all the macros to make C look like a special purpose programming language could be kept in a file named "macs". When a programmer wants to write in the new "macs language", he just writes the statement:

```
include macs
```

That single statement is the same as including all the macro definitions in his source file — but is much easier to type!

"Include" may also be used with structure definitions and other data descriptions. All of the definitions used for a given project may be bound together into several "include files" and then used in all of the sources that need them. When it comes time to change the data definitions, only the "include files" need be edited — not every source that uses them! This not only makes programming easier, it makes long term support by people not directly involved in the development of a project much simpler.

I/O AND C

Eventhough C is not defined with any intrinsic I/O statements, it still has a powerful I/O library. This library, called the "portable I/O library" is available in most environments which support C.

The I/O library provides for stream and record type I/O. The stream I/O is a unique concept for the HP/1000. It allows programmers to think of files as being long streams of bytes rather than records. The basic routines "getc()" and "putc()" allow the reading and writing of bytes of data to and from streams. Many other routines are available to open, close, create and purge streams.

Streams may be either standard FMGR files, or they may be logical units. The way they are defined may be changed at run time by the open function. The open function may be passed either a logical unit number or a string representing the namr of a file. After open is called, streams on logical units and streams on files behave the same way.

C also provides formatted I/O via the "printf()" and "scanf()" routines. This feature may be directed either to streams, or to memory to supply "CALL CODE" types of formatting.

The output routine "printf()" is powerful in the formatting capabilities it provides. Not only does it provide the FORTRAN type of functions, but it also give control over left and right field justification, zero or blank fill, and variable or fixed field widths.

"Scanf()" is the input routine and it provides many interesting input formatting features not found elsewhere. For example there are 3 different ways to input a string of characters:

- **Normal mode** — this is much like the "A" specifier in FORTRAN
- **Set mode** — Characters are read while they are members of a specified set. The specification %[0123456789] inputs a string as long as it is an integer (0 through 9).
- **Not set mode** — Characters are read while they are not members of a specified set. The specification %[^0123456789] inputs a string as long as it is not an integer (0 through 9).

The formatting routines can process 16, 32 and 64 bit quantities They have specifications for string, decimal, octal unsigned and even hexidecimal input and output.

In addition to all of these capabilities, C programs can elect to do their I/O processing directly through calls to FMGR and EXEC since the compiler interfaces to these routines so easily.

If your programs are very tight on space, you can even run the C formatter routines in a "stripped mode." All of the formatting routines ultimately call "putch()" and "getch()", so these may be replaced by specialized versions (which don't have the power of the supplied versions) which are much smaller. A simple version of put byte routine "putch()" could be coded as:

```
HPC,L,"PUTCH,7 TEST PUTCHAR";

extern exec();
int stdout;

putch(c)
char c;
{
  static char buf[100];
  static int i;

  if(c=='\n') {
    exec(2, stdout *(int [])buf, -i)
    i=0;
  }
  else buf[i++] = c;
}
```

As a final note on I/O, routines are available to randomly position streams to a specified byte. These are very useful for data base programs.

ASSEMBLY LANGUAGE

It may seem odd after talking about all the great features of the high level low level C language to mention assembly language. Indeed, for 90% of all applications we would anticipate no use for assembly language -- but there is still that nagging 10%!

HP/C addresses this problem with a solution which is quite portable. HP/C allows the programmer to embed assembly language in the C source code. This, in itself, is not at all portable and if abused will guarantee that no HP/C programs will ever be able to be compiled by any other C compiler. But if used carefully, embedded assembly language can be surprisingly portable. How?

Instead of embedding the assembly language directly into the C source we can make use of the macro pre-processor. We can define macros in terms of assembly language. The macros look so much like functions that they can be replaced by functions if the code ever has to be moved to another system. The result is portable C code, with assembly language.

Where would assembly language ever be used? Well for one thing some of the instructions of the HP/1000 are not found in all machines in the family. So assembly language may be used to implement some super fast routines which use instructions found only in some family members.

Assembly language may be used to interface the compiler to user microcode routines, or to interface to subroutines which don't use the standard .ENTR calling sequences. The vector instruction set would also be a good candidate for assembly language defined macros.

We will present here a very simple example of a "move string" macro implemented with assembly language.

```
HPC,1,p,"ASMBL Asmb test";

#define move(a,b,l) asm{ \
lda b; \
ldb a; \
mbt =l; }

#define SIZE 10

main()
{
char x[SIZE], y[SIZE];

  move(x,y, SIZE);
}
```

Note that the definition of the macro extends over several lines. The line extension in macro definitions is done through the use of the \ character.

The move macro as shown will expand to be:

```
LDA  Y
LDB  X
MBT  =B12
```

This code is the most rapid way to execute this procedure, and may be included directly in the C language source code.

By using the macro replacement assembly language code, C can interface to almost anything. Drivers may be written in C also (think of the uses of structures there!). Any previously implemented system can be easily interfaced, and most importantly, the interface is easily understood.

CONCLUSION

We have presented some of the many features which make the C language the programming language of choice on the HP/1000. There are many more features which make the C language the most attractive alternative to assembly language available.

HP/C was developed by Corporate Computer Systems under contract with Hewlett Packard. It is sold jointly by CCS and HP through a division of markets. CCS has granted HP an exclusive right to sell directly to the Bell System and to the U. S. Federal Government. CCS sells the compiler to all other domestic markets and offers support and enhancements to all market places.

The cost of the compiler on the first CPU is $4,000 while additional compilers are discounted to $2,000.

Two levels of support are available at $50 and $150 a month. The first level provides an automatic update service while the second provides the services of the first, along with phone in consulting.

CCS has also developed 2 courses on the C programming language which are given at various times during the year.

For further information call or write to: Corporate Computer Systems, Inc.
675 Line Road
Aberdeen, New Jersey 07747
201-583-4422

A NEW HP-IB COURSE

Bill Bolher/HP Data Systems Division

Due to popular demand, the "HP-IB in a Minicomputer Environment" course is being replaced by a new and improved course, cleverly titled "HP-IB in an HP-1000 Environment". The course materials include a new student workbook which was written and organized to reflect the many ideas, suggestions, and requests received from the field. Course topics include:

- HP-IB hardware and software overview
- RTE I/O and Bus addressing
- Introduction to Real-Time BASIC
- Programming the Bus and Bus devices
- Use of device subroutines
- HP-IB generaton and system considerations

Besides the student workbook, the students will also receive information on the HP-IB driver (DVR37), the Bus Interface, and applications for some popular Bus devices. The course is four days long and places increased emphasis on labs. Lecture and lab time is divided roughly 50-50. Employing the learning module concept, the labs have been written and organized so that rather than being strictly programming exercises, they require the students to refer to several HP-IB information manuals and answer typical HP-IB questions in addition to performing programming activities. Persons interested in, or planning on enrolling in the course, should be aware that a working knowledge of the RTE-IV operating system commands, as well as some familiarity with programming in Fortran and/or BASIC, is required if the student is to successfully attain the course objectives. Contact your local HP sales office for enrollment and course availability information.

A NEW TOOL FOR THE UNDERSTANDING OF RTE INTERNALS - RTE LISTINGS

Van Diehl/HP Data Systems Division

The long awaited RTE-IVB listings are now available!....

These source listings are available now for users interested in an understanding of the internals of the RTE-IVB operating system. These listings can be ordered in 800 bpi and 1600 bpi mag tapes, (not printed line printer copies, because of the enormous amount of paper required.) This way the user can easily print just the desired section of code, using a convenient directory provided in the beginning of the tape.

The RTE-IVB assembler listings are provided for the benefit of the system programmer that wants to acquire a knowledge of RTE internals operation. They should be invaluable additional material for users that have attended the RTE Internals Course — ADVANCED RTE WORKSHOP.

These listings are not to be confused with the RTE-IVB source product — 92068X — which contains all sources of the product, and is intended for users who desire to modify the operating system.

The RTE-IVB listings have been obtained by assembling parts of the operating system that are more frequently used. The parts included are:

1. The Memory Resident System — containing the scheduler, dispatcher, session monitor, real time I/O control module, etc.
2. Utilities — containing operating system utilities such as KEYS, KYDMP, WHZAT, READT, WRITT, SAVE, RSTOR, COPY, etc.
3. I/O Drivers — contains the code for 16 I/O drivers for terminals, line printers, mag tape, discs, card reader, etc.
4. Decimal String Arithmetic Routines
5. HP-IB Library
6. TV Interface Card Library — a graphic library to draw characters, vectors and points on a TV monitor
7. System Library
8. Disc Backup Library
9. DOS/RTE Relocatable Library — containing arithmetic, transcendental functions, etc.

ORDERING INFORMATION

The source listings should be ordered by the following part numbers:

- Source Listings in 800 bpi mag tape — 22999-90236. List price: $I500.00
- Source Listings in 1600 bpi mag tape — 22999-90237. List price: $1500.00

The availability of these listings are 4 weeks from receipt of the order.

These listings are for RTE-IVB date code 1940 (10/1/79) and will be updated at approximately six month intervals. No update service will be provided.

For internal orders please note that it is imperative that they be placed using a type I2 HEART order, on division 2200, ID code 42. Customer orders should be placed as usual on the local sales office and transmitted via HEART on supplying division 2200.

PRESENT A PAPER AT THE FIRST HP/1000 INTERNATIONAL USERS GROUP CONFERENCE!

The HP/1000 International Users Group is now proceeding to make plans for the First Annual HP/1000 International Users Group Conference to be held in San Jose, California, in August 1980. Following is a reprint of the call for papers to be presented at the conference. For more information, contact Dr. Glen Mortensen at the address given below.

It's not too late to start finalizing your papers on your nifty HP/1000 applications!

CALL FOR PAPERS
HP/1000 INTERNATIONAL USERS GROUP
1980 CONFERENCE
"LET'S GET STARTED"

The theme of the first annual conference is intended to stimulate communication among HP/1000 users and to survey the varied uses of HP software/hardware in the field. This includes, but is not limited to, end users and OEM's worldwide who feel they can bring some information of general interest to the conference and to exchange this information in an open forum with other HP/1000 users.

The 1980 conference will be held in San Jose, California, on August 25, 26, and 27, 1980. Submissions are invited from all interested parties.

Each submission should consist of both an abstract and a summary. Abstracts must not exceed 50 words in length and will be published with the conference agenda. Summaries must not exceed 1000 words in length and must include a statement of the need, method(s) of solution, and conclusions. Supporting graphics may be included if desired. A suggested format for submission is as follows:

IDENTIFYING INFORMATION

- Category (see below)
- Title of Paper
- Author's Name
- Affiliation
- Address

ABSTRACT (maximum of 50 words)

SUMMARY (maximum of 1000 words)

- Statement of Need
- Method(s) of Solution
- Conclusion

Approval for release of the paper should be obtained prior to submission of abstracts and summaries. Material submitted for consideration will not be returned. Six (6) copies of each submission should be forwarded for review to:

GLEN A. MORTENSEN
Intermountain Technologies, Inc.
P. O. Box 1604
Idaho Falls, ID 83401

no later than April 1, 1980. Authors will be notified of acceptance by June 1, 1980. Submission of full papers in final format will be required by August 1, 1980.

Some suggested categories for papers are listed below. This list is in no way intended to be all-inclusive, and papers in other subject areas are welcomed and encouraged.

1. COMPUTATIONAL APPLICATIONS

- Graphics
- Microprogramming
- CAD/CAM
- Word Processing
- Interactive Applications
- Other Specific Applications

2. DATA COMMUNICATIONS

- Remote Job Entry
- Distributed Processing
- Data Communication Drivers
- X.25 Standard
- Multiplexers
- Interfacing Alien Devices/Systems
- System Network Architecture

3. INSTRUMENTATION

- IEEE 488/HP-IB
- Process Control
- Data Acquisition
- Supervisory Control
- Instrument Interfaces

4. OPERATING SYSTEMS

- RTE
- Drivers for RTE
- Languages
- File Management Package

5. OPERATIONS MANAGEMENT

- Factory Data Collection Systems
- Data Base Management
- Resource Management (Software, Hardware, People)
- Resource Optimization

PROGRAM OPTIMIZATION

Mike Manley/DSD Lab

Hewlett-Packard is pleased to announce the addition of a new product to its HP 1000 product line. The product is called ACCEL/1000. ACCEL/1000 is a software package that will give an execution time profile of any program that runs under RTE control with an elapsed time of at least a few seconds. The program is an extremely powerful tool that was specifically designed to allow you to fine tune and optimize your application programs.

Optimization of program execution time is the focus of much effort throughout all sections of the computer industry. In areas such as sorting, searching, data acquisition, terminal response, and many others, the entire worth of a program or product is judged solely on the speed with which it can perform a given task. Programs that execute faster perform better on benchmarks and typically sell better than slower, more inefficient competitors. Programs that execute faster take less machine time, and thus leave more of the machine for other processes. In the data processing industry the introduction of a faster more efficient piece of software to perform a given task has been known to make or break entire software houses.

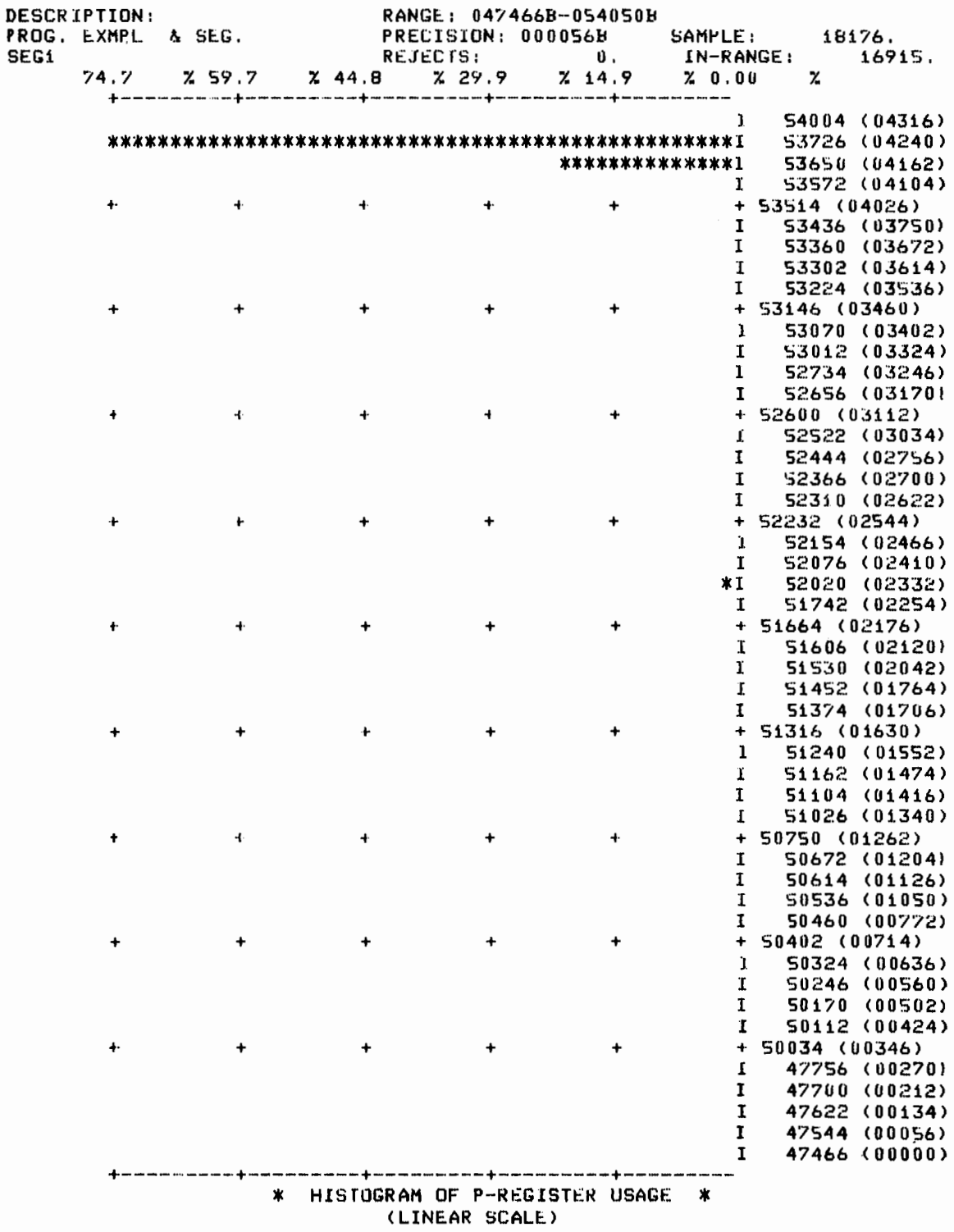
One of the interesting things about program optimization is that in the vast majority of cases, 75% to 95% of program execution time is spent in 3% to 7% of the code. For example, in the RTE IV relocating loader (LOADR) 26% of execution time is spent in a four word loop looking for an existing base page link to reuse. Another 27% of the LOADR's time is spent in five words doing symbol table comparisons. Before using ACCEL/1000 on the RTE IV relocating loader, the base page loop was twelve words long and the symbol table comparison was nine words long. After using ACCEL/1000 on the relocating loader, these and a few other critical areas were pinpointed. The code was changed in these areas and as a result the LOADR ran 61% faster.

The trick, however, is not changing the code. That's easy. The trick is finding what area of code should be changed. In most cases this is not obvious, especially in large segmented programs. It does no good to optimize a portion of code that is only executed 1% of the time when the same effort could be applied to an area of code executed 50% to 80% of the time. Moreover, if the application is extremely time critical it would be useful to know which area of a program would be best to micro code. Obviously, the area to micro code is the area where the program spends the most time.

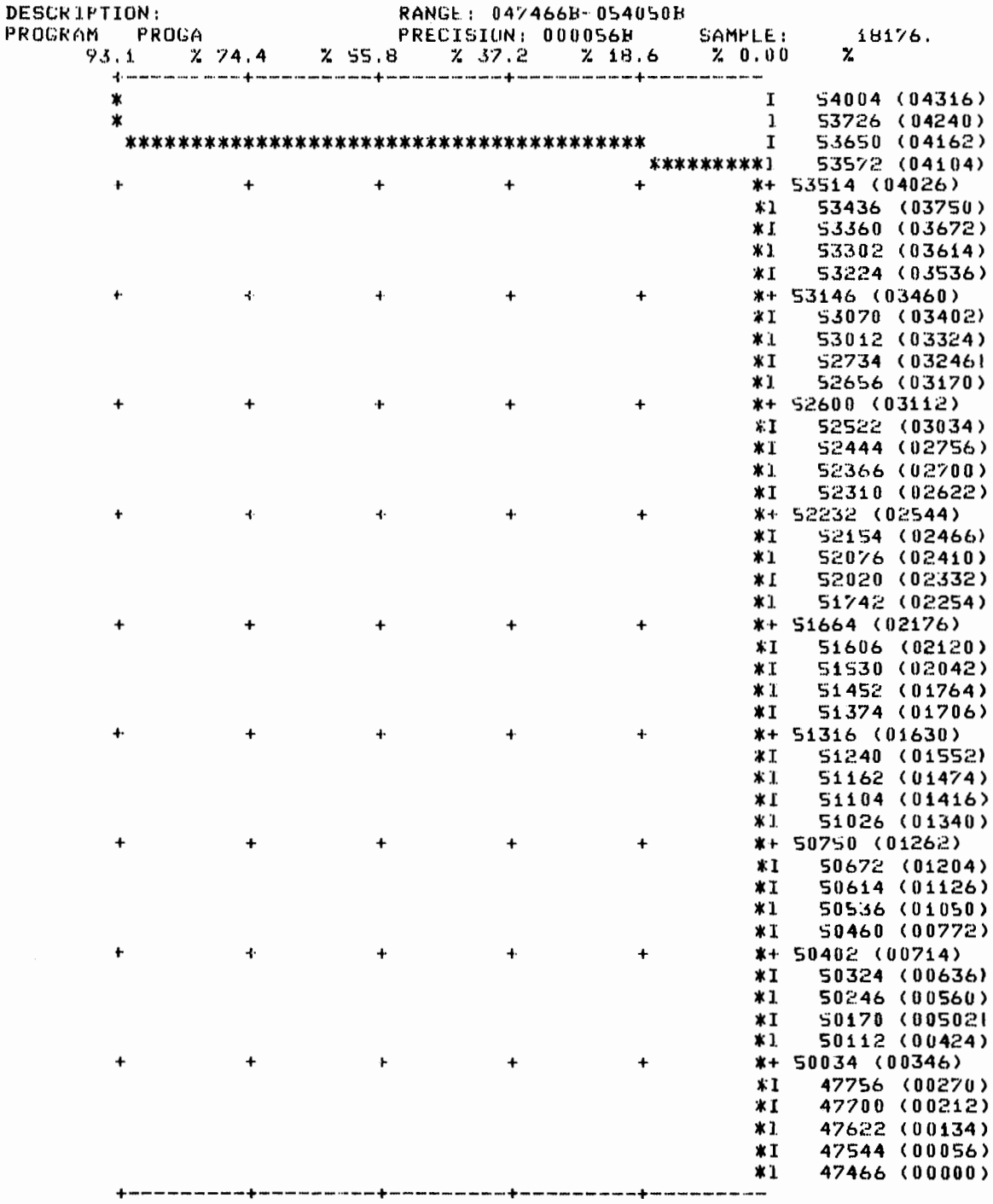
The ACCEL/1000 package will tell the user exactly where to spend his time on optimization. This optimizing and profiling package is a collection of software that will allow the user to automatically generate execution time profiles of any program that will execute under the RTE IV operating system. Programs running under this package may be segmented, background, real time, or any combination of these. No program code changes are required to run the package, and the package will run on any M series, E series, or F series CPU. The profiler also requires no special hardware or micro code. The programs are conversational, interactive, and so easy to use that no system expertise is required. Moreover, the program to be monitored may be written in any language.

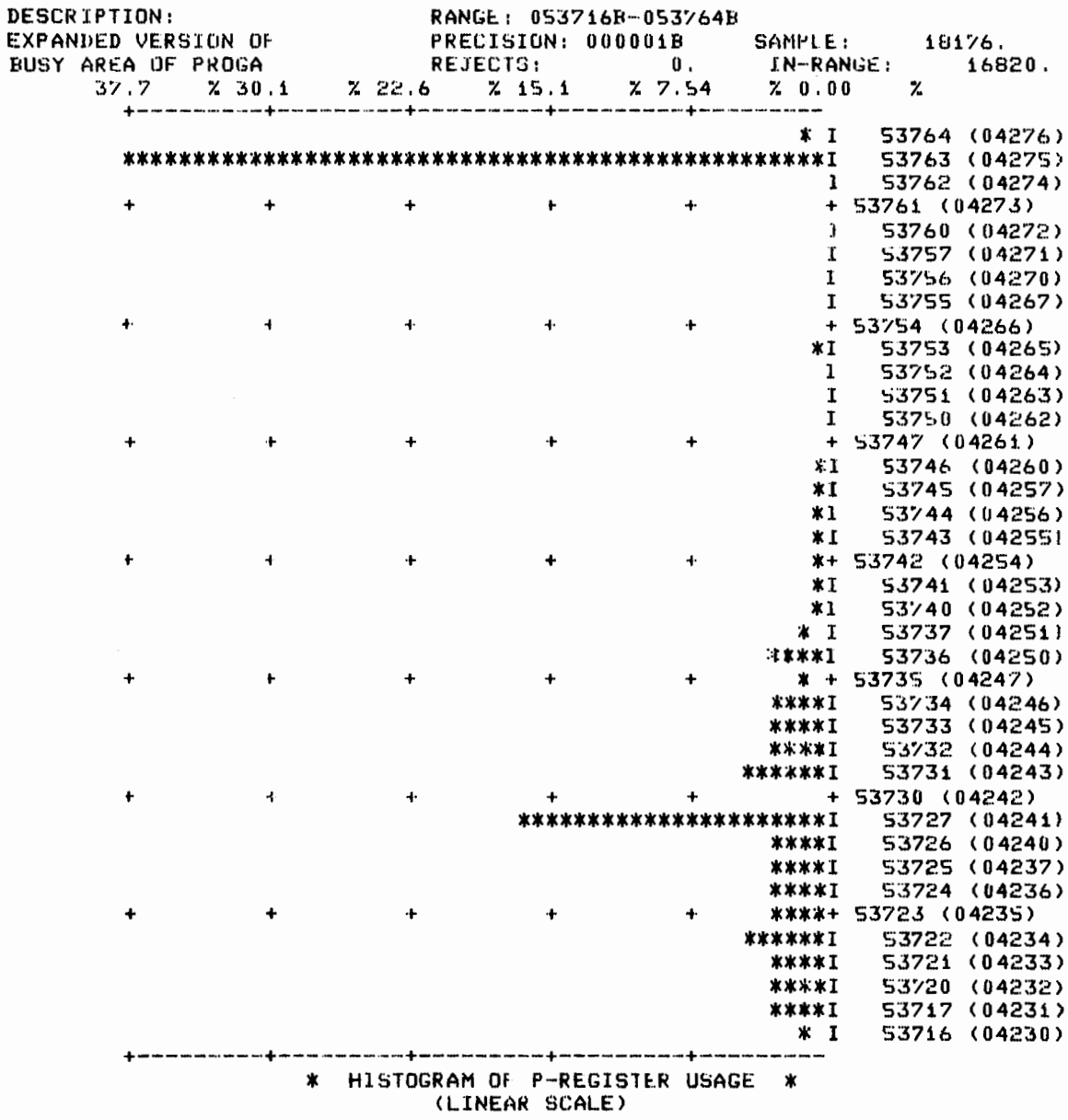
During the time the target program is run under profiler control, data concerning the profile of that program is kept on the disc. When the program has completed the data file is closed and another section of the package may be invoked to print out the profile results. The target program may be analyzed even down to the individual instruction level.

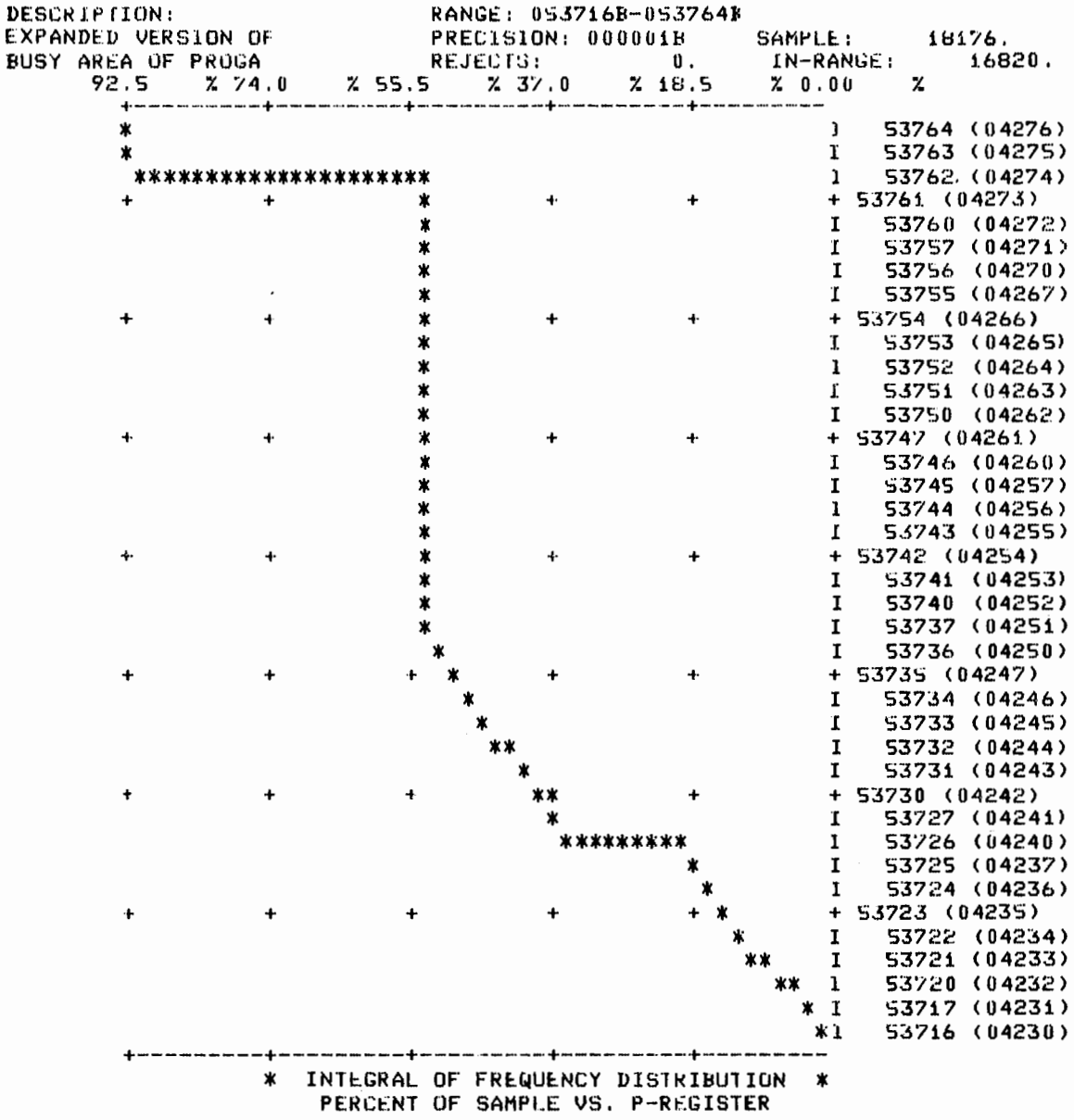
Two examples of the results of profiling a program are shown on the next page. The first pair of graphs show a coarse overview of the entire program giving a histogram of P-register usage and an integral of that histogram. The second pair of graphs show the profiler's ability to zoom in on the areas of interest. Note that the second example gives information at the individual instruction level. Lastly, to give you an idea of how useful we think this package really is, it is our own lab's policy not to release any software until that software has been extensively profiled with ACCEL/1000.



* HISTOGRAM OF P-REGISTER USAGE *
(LINEAR SCALE)


DESCRIPTION:
RANGE: 047466B-054050B
PROGRAM PROGA
PRECISION: 000056B
SAMPLE: 18176.
93.1 % 74.4 % 55.8 % 37.2 % 18.6 % 0.00 %
53514 (04026)
53436 (03750)
53360 (03672)
53302 (03614)
53224 (03536)
53146 (03460)
53070 (03402)
53012 (03324)
52734 (03246)
52656 (03170)
52600 (03112)
52522 (03034)
52444 (02756)
52366 (02700)
52310 (02622)
52232 (02544)
52154 (02466)
52076 (02410)
52020 (02332)
51742 (02254)
51664 (02176)
51606 (02120)
51530 (02042)
51452 (01764)
51374 (01706)
51316 (01630)
51240 (01552)
51162 (01474)
51104 (01416)
51026 (01340)
50750 (01262)
50672 (01204)
50614 (01126)
50536 (01050)
50460 (00772)
50402 (00714)
50324 (00636)
50246 (00560)
50170 (00502)
50112 (00424)
50034 (00346)
47756 (00270)
47700 (00212)
47622 (00134)
47544 (00056)
47466 (00000)
INTEGRAL OF FREQUENCY DISTRIBUTION
PERCENT OF SAMPLE VS. P-REGISTER


DESCRIPTION:
EXPANDED VERSION OF
BUSY AREA OF PROGA
RANGE: 053716B-053764B
PRECISION: 000001B
REJECTS: 0.
SAMPLE: 18176.
IN-RANGE: 16820.
37.7 % 30.1 % 22.6 % 15.1 % 7.54 % 0.00 %
53764 (04276)
53763 (04275)
53762 (04274)
53761 (04273)
53760 (04272)
53757 (04271)
53756 (04270)
53755 (04267)
53754 (04266)
53753 (04265)
53752 (04264)
53751 (04263)
53750 (04262)
53747 (04261)
53746 (04260)
53745 (04257)
53744 (04256)
53743 (04255)
53742 (04254)
53741 (04253)
53740 (04252)
53737 (04251)
53736 (04250)
53735 (04247)
53734 (04246)
53733 (04245)
53732 (04244)
53731 (04243)
53730 (04242)
53727 (04241)
53726 (04240)
53725 (04237)
53724 (04236)
53723 (04235)
53722 (04234)
53721 (04233)
53720 (04232)
53717 (04231)
53716 (04230)
* HISTOGRAM OF P-REGISTER USAGE *
(LINEAR SCALE)


DESCRIPTION:
EXPANDED VERSION OF
BUSY AREA OF PROGA
RANGE: 053716B-053764B
PRECISION: 000001B
REJECTS: 0.
SAMPLE: 18176.
IN-RANGE: 16820.
92.5 % 74.0 % 55.5 % 37.0 % 18.5 % 0.00 %
53764 (04276)
53763 (04275)
53762 (04274)
53761 (04273)
53760 (04272)
53757 (04271)
53756 (04270)
53755 (04267)
53754 (04266)
53753 (04265)
53752 (04264)
53751 (04263)
53750 (04262)
53747 (04261)
53746 (04260)
53745 (04257)
53744 (04256)
53743 (04255)
53742 (04254)
53741 (04253)
53740 (04252)
53737 (04251)
53736 (04250)
53735 (04247)
53734 (04246)
53733 (04245)
53732 (04244)
53731 (04243)
53730 (04242)
53727 (04241)
53726 (04240)
53725 (04237)
53724 (04236)
53723 (04235)
53722 (04234)
53721 (04233)
53720 (04232)
53717 (04231)
53716 (04230)
INTEGRAL OF FREQUENCY DISTRIBUTION
PERCENT OF SAMPLE VS. P-REGISTER

JOIN AN HP 1000 USER GROUP!

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

NORTH AMERICAN HP 1000 USER GROUPS

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

NORTH AMERICAN HP 1000 USER GROUPS (CONTINUED)

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

OVERSEAS HP 1000 USER GROUPS

London	Rob Porter Hewlett-Packard Ltd. King Street Lane Winnersh, Workingham Berkshire, RG11 5AR England (734) 784 774
Amsterdam	Mr. Van Puten Institute of Public Health Anthony Van Leeuwenhoeklaan 9 Postbus 1 3720 BA Bilthoven The Netherlands
South Africa	Andrew Penny Hewlett-Packard South Africa Pty. private bag Wendywood Sandton, 2144 South Africa

THE PLUS/1000 LIBRARY NOW HAS A HOME!

All correspondence should be directed to

Dave Olson
PLUS/1000 Librarian
1846 W. Eddy St.
Chicago, IL 60657

All phone inquiries should be made to:

(312) 525-0519

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

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

Printed in U.S.A. 3/80

Part No. 5951-6111