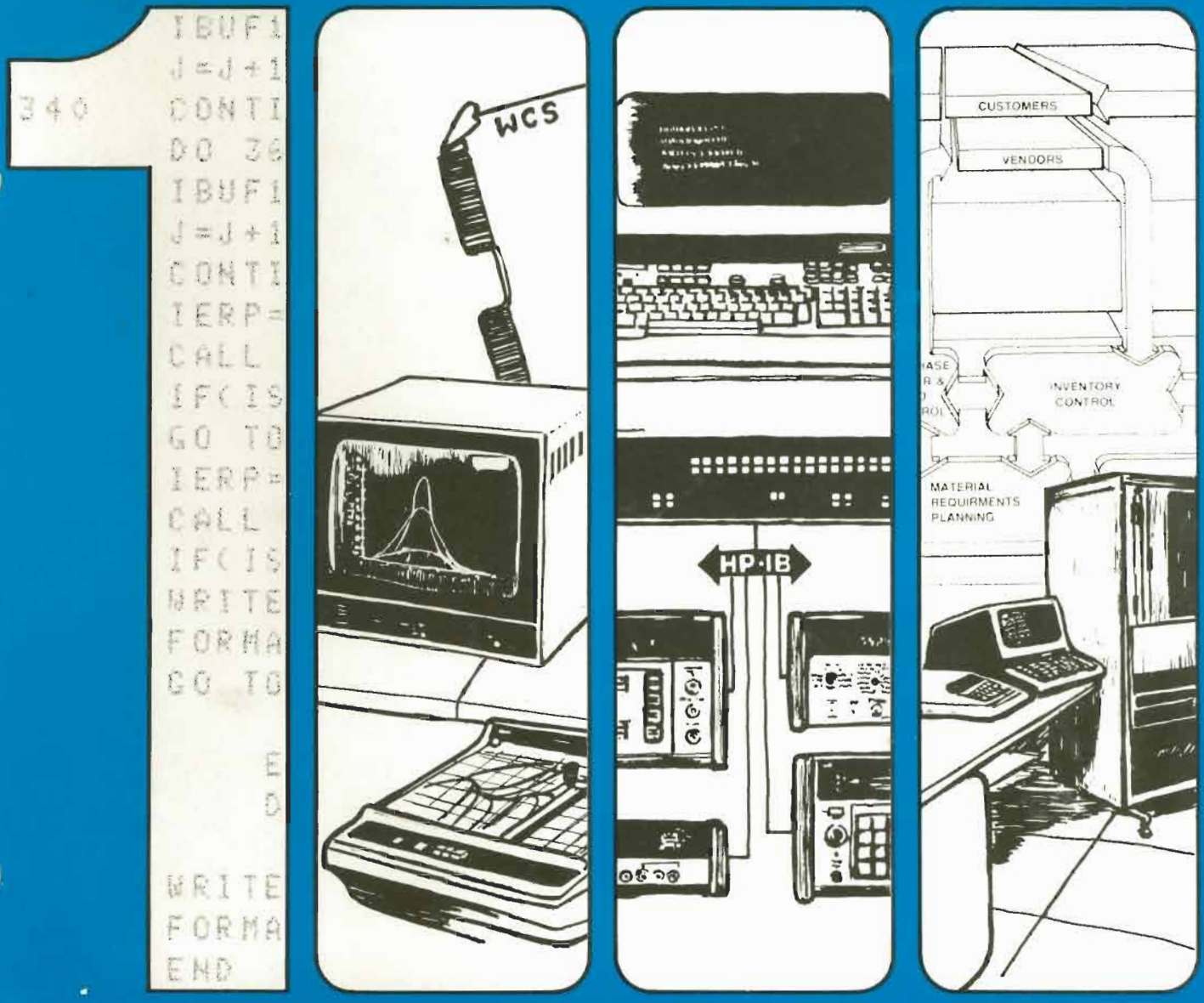
HEWLETT PACKARD

Hewlett-Packard Computer Systems

COMMUNICATOR



1981

Volume V Issue 1

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HEWLETT-PACKARD
COMPUTER SYSTEMS

Volume V
Issue 1

COMMUNICATOR/1000

Feature Articles

Departments

ABOUT THIS ISSUE

The first issue of the Communicator/1000 for 1981 could be titled the "HP Employee" issue. Most articles published were submitted by either HP Field personnel or Division employees. This is because we haven't been receiving many articles from HP customers.

Readers please note that articles for future issues of the Communicator/1000 have not yet been selected. If you have found a super-efficient or tricky way to use an HP 1000 why not write an article, and let the 2600 Communicator readers share your excitement. Remember each issue awards HP32E Calculators to feature article contributors from each catagory (HP 1000 customers, HP Field employees, and HP Division employees).

In this issue we have two articles in the area of operating systems. Kent Ferson has written a very useful article for RTE-L users. Kent outlines the advantages and describes the mechanics of sharing the Fortran Formatter between programs. Unfortunately Kent wasn't eligible for the calculator award because he is an employee of Data Systems Division's Technical Marketing Department. The other operating system article was contributed by Larry Smith of Manufacturing and Consulting Services Santa Anna, California. Larry has written other articles which have been published in the Communicator/1000, and "Sharing Physical Memory in Fortran and Pascal" is another good one.

In the Languages Category we have a super article written by John Stafford of HP's Data Systems Division. John has compared HP's Pascal/1000 to the UCSD Pascal, which has become popular in the small computer area. John's well written article is intended to point out the differences between these two implementations of Pascal.

HP's Carl Reynolds of Rochester New York has contributed an article on Datacap/1000 for the Operations Management Section. Carl's experience designing a Datacap/1000 system has enabled him to recount his expertise in "Designing a High Performance Datacap/1000 System."

Since this time there was only one article in each category, each one of the feature article authors wins the HP32E calculator!!!

Best feature article by an HP Customer:

Larry B. Smith Manufacturing and Consulting Services Inc.	"Accessing Physical Memory in Fortran and Pascal"

Best feature artice by an HP Division employee not in Technical Marketing:

John Stafford HP Data Systems Division	"A Comparison of Pascal/1000 to UCSD Pascal"

Best feature article by an HP Field Employee:

Carl Reynolds HP Rochester, New York	"Designing a High Performance Data Capture Sysytem"

I hope you enjoy reading the articles in this issue and continue to provide the Communicator/1000 with interesting and useful material.

Sincerely,

The Editor

EDITOR'S DESK

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

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

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

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

WIN AN HP-32E CALCULATOR!

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

1. HP 1000 Customers;

2. HP field employees;

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

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

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

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

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

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

   OPERATING SYSTEMS
   DATA COMMUNICATIONS
   INSTRUMENTATION
   COMPUTATION
   OPERATIONS MANAGEMENT

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

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

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

A SPECIAL DEAL IN THE OEM CORNER

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

IF YOU'RE PRESSED FOR TIME . . .

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

THE MECHANICS OF SUBMITTING AN ARTICLE

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

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

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

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

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

With best regards,

The Editor

EDITOR'S DESK

LETTERS TO THE EDITOR

Dear Editor,

I would like to publish a correction to my Bit Bucket article, "The SST in the Session Control Block," (Vol IV, Issue 4). A LOGON 09 error occurs if the maximum size of the SST is greater than 70, not 63.

Sincerely,

Martha Slettedahl
HP Data Systems Division
Cupertino, CA.

Dear Martha,

Thank you for notifying us of this correction.

Sincerely,

The Editor

Dear Editor,

Referring to page 15, entitled "Put Your 264X Terminal Display on Paper", in Communicator Volume IV Issue 3, a more novel approach exists in the HP 1000 Users Group Library. The program is called SCAN. Listed here is my modified version of the library program SCAN. It has an important feature allowing a copy of memory from cursor position to the command string :RU,SCAN. Therefore, it is not necessary to dump the whole memory or pre-edit it prior to making a hard copy.

Sincerely,

Charles L. Elliot
Elliot Geophysical Company

Dear Charles,

Thanks for pointing out that the HP 1000 User's Group contributed library contains many useful programs. More information on programs in the contributed library can be obtained from the HP 1000 User's group contacts. The Bulletin section of this Communicator contains a list of HP 1000 User's Groups.

Sincerely,

The Editor

```
FTN4,L
C***********
C#0176 SCAN
C***********
      PROGRAM SCAN(3),791121 C.L. ELLIOT
C     ADAPTED FROM LOCUS PROGRAM 22683-90017
C     THIS PROGRAM EMULATES A HARD COPY DEVICE BY
C     READING THE CONTENTS OF THE 26XX MEMORY
C     AND WRITING THAT DAT TO LUPR
C     DEFAULT IF LUPR=6
C     RU,SCAN,LUPR WILL COY FROM CURSOR POSITION
C     AND ENDS AT ^^^^ OR :RU,SCAN
C     DIMENSION IBUF(50),IPAR(5),IREG(2)
C     EQUIVALENCE (REG,IREG(1),IA),(IREG)(2),IB)
C     GET TERMINAL LOGICAL UNIT NUMBER AND PARAMETERS
      CALL RMPAR(IPAR)
      LU=LOGLU(LU)
      LUN=LU
      IOUT=IPAR(1)
      IF(IPAR(1).EQ.LU)IOUT=6
C     DEFINE CONTROL FUNCTIONS
      IDC1=10537B
      NLESC=33B
      IDUN=62137B
      IUP=40537B
      IEND=57136B
    5 FORMAT(2A2)
C     PUT TERMINAL IN BLOCK MODE
      WRITE(LU,5)IDC1
      WRITE(LU,5)NLESC,IUP
C     REQUST 1 LINE FROM TERMINAL
   20 WRITE(LU,5)NLESC,IDUN
C     READ THE LINE
      IBUF(1)=2H
      REG= EXEC(1,LUN,IBUF,50)
      LEN=IB
C     CHECK FOR END -- ^^^^ OR :RU,SCAN
      IF(LEN.GT.6) GO TO 50
      IF((IBUF(1).EQ.IEND).AND.(IBUF(2).EQ.IEND))GOTO99
      IF((IBUF(1).EQ.2H:R).AND.(IBUF(2).EQ.2HU,)
     $.AND.(IBUF(3).EQ.2HSC).AND.(IBUF(4).EQ.2HAN))GO TO 99
   12 FORMAT(1H ,50A2)
C     WRITE THE LINE TO THE LINE PRINTER
   50 WRITE(IOUT,12)(IBUF(J),JJ=1,LEN)
      GO TO 20
C     EJECT PAGE AND END
   99 ICNWD=IOR(1100B,IOUT)
      CALL EXEC(3,ICNWD,-1)
  999 END
      END$
```

Dear Editor,

To be able to copy information from the 264X display to a line printer is indeed handy. The program PSCRN by David Liu of Hughes Aircraft Company (Communicator/1000, Volume IV, Issue 3, pages 15, 16 and 17) was installed on our RTE-IVB/ Session Monitor system and ran fine — until the new EDIT/1000 RTE Editor was run.

As you will recall, Mr. Liu's article says (bottom of page 15) that the D switch on the 02640-60123 board must be set to open. Making that switch change inside the terminal posed no particular problem. PSCRN ran great. However, when EDIT/1000 was run, the following unexpected message appeared on the CRT along with a resounding bell: "Setting terminal straps to d g l T". The Editor had programatically closed the D switch (d)!

Thereafter, the PSCRN program would work only if the D switch was reset to open by a HARD terminal reset. Therein lies the "gotcha". A hard reset also clears the CRT of the information destined for the line printer.

A few minutes spent with the 264X Terminal Reference Manual (02645-90005), pages 3-1 and 7-19 revealed the technique for setting the D switch programatically with an escape sequence. A few statements added to Mr. Liu's FORTRAN program now allows the D switch to remain in its normal closed position; the CRT-to-line printer copying now peacefully co-exists with EDIT/1000.

The partial listing shows the statements added. See lines 26, 27, 59, 60, 97 and 142.

If one felt the need to accomodate CRT configurations where the D switch was to be left in its orginal state, whichever it might be, additional calls could be made to determine the terminals primary status (see pages 6-2 and 6-3 of the Reference Manual), and then program/reprogram the terminal only as appropriate.

Thanks, Mr. Liu for your article. It certainly shows another of the flexible features of Hewlett-Packard terminals.

Sincerely,

Ray Tatman
HP Instrument Service Training Center

Changed lines

```
26        INTEGER PGMODE (3)
27        INTEGER LNMODE (3)
59        DATA PGMODE /2H%&,2H=1,2HD /
60        DATA LNMODE /2H%&,2H=0,2HD /
97        REG = EXEC(2,1400B+CRT,PGMODE,-5,PARM1,PARM2)
142       REG =EXEC(2,1400B+CRT,LNMODE,-5,PARM1,PARM2)
```

Dear Ray,

Thank you for letting us know this enhancement to the PSCRN program. I'm sure that the users of EDIT/1000 would be interested. Rather than reprint the whole program I'm just going to list the changed lines.

Sincerely,

The Editor

EDITOR'S DESK

Dear Editor,

In my article "Fast Fortran" there was a mistake in one of the examples. The original was as follows:

```
CALL OUT ("CURRENT LU IS      ", LUOUT,10,LU)
```

It should read:

```
CALL OUT (14HCURRENT LU IS , LUOUT, 10,LU)
  .
  .
  .
END

SUBROUTINE OUT (IBUF, IVAL, N, LU)
INTEGER IBUF(1), OBUF(10)
DO 10 I=1, N-2
10 OBUF(I)=IBUF(I)
CALL CNUMD (IVAL, OBUF(N-2))
CALL EXEC (2,LU,OBUF,N)
RETURN
```

WHERE

N = −# of words (not characters) in passed array plus 3 more words for IVAL.

IVAL = Value to be printed after message.

OBUF = Local array big enough to hold largest N words.

LU = Print logical unit.

Note: In main program, arrays IRBUF and IBUF must be dimensioned (as 33 and 3 for this example).
Thanks for the calculator.

Best regards,

John A. Pezzano
HP, El Paso

Dear John,

Thank you for notifying us of this error in vol. 4 issue 4, and for writing a good article!

Sincerely,

The Editor

Dear Editor,

With regard to the recent article entitled "Generating RTE for Pleasure and Performance," take care when deleting %CLIB from the library section of the program input phase. BASIC needs %BASLB at gen-time which has external references to %CLIB! The other library modules mentioned may be deleted as shown, however, the big time savings on generation time is probably lost with the inclusion of %CLIB. Sorry 'bout that...

Sincerely,

Ron Hammer
HP, Saint Louis

MEASURING TIME BASE GENERATOR OVERHEAD

Bob Hallenbeck/HP San Diego

The following is a program to measure the amount of time an HP operating system spends in servicing interrupts from the time base generator. The time spent is expressed as a percentage of overhead.

First, a brief description of TBG interrupt servicing. When an interrupt is received from the TBG, it is vectored through $CIC in the usual manner. The interrupt servicing routine first increments the system time word, then scans the time list to see if a program should be scheduled, then checks each of the EQT entries to see if a time out has occurred. If a time out has occurred, then either the driver continuation section is entered, or the LU will be set down and a time out message will be output.

Thus, the time it takes to service a TBG interrupt depends upon the number of programs in the time list at the time of the interrupt, and upon the number of EQT entries in the system. For the sake of accuracy it is assumed that there will be no entries in the time list, and no time outs occurring at the time the overhead program is run.

The program is independent of CPU and operating system. It consists of a FORTRAN main program (TBGOH) to collect statistics, a FORTRAN subroutine (FRMAT) to format the results, and an assembly language subroutine (TBG) to program the TBG. For the measurement technique I am indebted to Jim Nissen (HP).

The idea of the main program is to program the TBG card to interrupt every 10 milleseconds (default interrupt frequency), enter a do nothing loop one million times, then calculate and save the indicated elapsed time. This procedure is then repeated with the TBG programmed to every millisecond (ten times faster), to observe the indicated elapsed time. Some simple algebra is then done on the indicated elapsed time, and a percentage overhead figure is obtained.

The assembly language subroutine (TBG) is used to program the time base generator. The reason for the rather indirect method of building the appropriate instructions is that the arguments for the LIA, OTA and STC instructions must be constant, but the TBG may be in almost any select code.

```
FTN4,L,Q
      PROGRAM TBGOH
      INTEGER STIME,FTIME,ETIME,PASS
      DIMENSION STIME(5),FTIME(5),ETIME(5),IPARM(5)
      EQUIVALENCE (IPARM(1),LU)
      DATA PASS/0/
      CALL RMPAR(IPARM)
C*********************************************************************
C
C         REL,%TBGOH
C         REL,%TBG
C         EN
C
C         RU,TBGOH,[LIST LU]
C
C*********************************************************************
C
C     PROGRAM THE TBG TO INTERRUPT EVERY 10 ms
C
      CALL TBG(2)
C
C     START TIMER
C
5     PASS=PASS+1
      CALL EXEC(11,STIME)
C
C     ENTER DO NOTHING LOOP - EXECUTE 1 MILLIOM TIMES
C
      DO 100 I=1,1000
      DO 100 J=1,1000
100   CONTINUE
C
C     STOP TIMER
C
      CALL EXEC(11,FTIME)
      IF (PASS.EQ.2)GOTO 200
C
C     CALCULATE AND SAVE ELAPSED TIME
C
      CALL FRMAT(STIME,FTIME,ET1)
      WRITE(LU,500)ET1
500   FORMAT(" ET1 = ",F12.3," SEC")
C
C     PROGRAM THE TBG TO INTERRUPT EVERY 1 MS
C
      CALL TBG(1)
      GO TO 5
C
C     RESTORE THE TBG TICK TO 10 MS
C
200   CALL TBG(2)
```

```
C
C     FORMAT THE ELAPSED TIME
C
      CALL FRMAT(STIME,FTIME,ET2)
      WRITE(LU,510)ET2
510   FORMAT("ET2 = ",F12.3," SEC")
C
C     CALCULATE  PERCENTAGE OVERHEAD
C       USING ET1 AND ET2
C
      OVHD = 10*(ET2 - 10*ET1)/(ET2-ET1)
      WRITE(LU,520)OVHD
520   FORMAT(" TBG OVERHEAD = ",F12.3,"%")
      END
C
C
C
      SUBROUTINE FRMAT(STIME,FTIME,ET)
      INTEGER STIME,FTIME,ETIME
      DIMENSION STIME(5),FTIME(5),ETIME(5)
      DO 100 I=1,4
      ETIME(I)= FTIME(I)-STIME(I)
100   CONTINUE
      IF(ETIME(4).LT.0)ETIME(4) = ETIME(4) +24.
      ET=ETIME(4)*3600. + ETIME(3)*60. + ETIME(2) + ETIME(1)/100.
      RETURN
      END
      END$
```

```
ASMB,L
      NAM TBG 3,10      PROGRAM TBG INTERUPT FREQUENCY
      ENT TBG
      EXT .ENTR,EXEC,$LIBR,$LIBX
*
CODE  BSS 1
TBG   NOP                       .ENTR SEQUENCE
      JSB .ENTR
      DEF CODE
*
      CLA                       PICK UP TBG SETTING (0 = .1 ms, 1 = 1 ms,
      LDA CODE,I                2 = 10 ms, 3 = 100 ms, 4 = 1 sec, 5 = 10 sec,
      STA TIME                  6 = 100 sec., 7 = 1000 sec.
*
      LDA .TBG,I                GET TBG SELECT CODE FROM BASE PAGE
      IOR LIA
      STA .LIA                  BUILD LIA SC INSTRUCTION
      ADA =B100
      STA .OTA                  BUILD OTA SC INSTRUCTION
      ADA =B1100
      STA .STC                  BUILD STC SC,C INSTRUCTION
*
      JSB $LIBR
      NOP
*
      LDA TIME
.OTA  OTA 10                    OUTPUT SETTING TO TBG CARD
.STC  STC 10,C                  START CLOCK AND CLEAR FLAG
.LIA  LIA 10                    BIT 4 SET IF ANY TBG TICKS MISSED
*
      JSB $LIBX
      DEF *+1
      DEF *+1
*
      JMP TBG,I
.TBG  OCT 1674
LIA   LIA 0
TIME  BSS 1
      END TBG
```

RS-232 LINK BETWEEN THE HP 1000 AND THE HP 85

By Bob Niekamp/HP Kansas City

Communication to the HP1000 from the HP85 using RS232 can be accomplished by using the programs listed in this article. This application describes how to send data between the HP85 and the HP1000. These programs are not terminal emulators for the HP85. Also, these programs are for use over a modem (Bell 212A) at 1200 baud, auto-answer or originate. To connect the HP85 directly to the HP1000, you could use a modem eliminator cable (13232U) between the 12966-60006 cable and the HP85 RS-232 interface cable. You must configure the 12966 interface card on the 1000 to 1200 baud, manual line open (CN,LU,31B,1B) for direct connection without modems. The programs on the HP85 are written to handshake with DVA05 on the HP1000 using enquire/acknowledge protocol. System generation configuration for DVA05 is the same as for HP264X terminals.

HARDWARE AND SOFTWARE REQUIREMENTS

HP85A with I/O ROM.

HP1000 E or F with 12966 interface card, 12966-60006 modem cable, RTE-IVB, DVA05, FTN4 compilier.

BELL 212A modem.

SET-UP

Execute the transfer file '/HP85' on the 1000 to map in the HP85 logical unit to your session, to set the 12966 card to 1200 baud auto-answer, and disable LOGON prompt on interrupt from the HP85.

USE

Program 'RECV' is run on the 1000 to accept data from the HP85. In conjunction with this, you would execute program 'XMT85' on the HP85 to send data to the HP1000. The procedure to send data from the HP85 to the HP1000 is as follows: run 'XMT85' on the HP85 to initialize the modem lines properly. The program will wait until the line is established (i.e. when the HP85 modem is called and it automatically answers or when you dial the HP1000 modem from the HP85.) At this point, you then run 'RECV' on the HP1000 to initialize the transfer. If handshaking does not occur properly, you should get an error message.

To send data from the HP1000 to the HP85, program 'XMIT' is run on the HP1000 to send data to the HP85 and program 'RCV85' is run on the HP85 to receive the data from the 1000. The procedure is the same as the one described above except run 'XMIT' on the 1000 in place of 'RECV' and run 'RCV85' on the HP85 in place of 'XMT85'. If the link hangs, then execute the transfer file '/HP85' which should clear things out.

There are two modem routines listed here, auto-answer and originate. Use the auto-answer module if you are automatically dailing the HP85 from the HP1000. Otherwise, use the originate routine to dial the HP1000 from the HP85.

```
:****TRANSFER FILE TO SET UP 12966A CARD
:****FOR 1200 BAUD, NO PROMPT & MAP HP85 LU INTO SESSION
:****HP85 LU IS 18.
:****
:SYLU,18,0
:SYLU,18,18
:CN,18,21B
:** SET UP 1200 BAUD AND NO PARITY
:CN,18,30B,411B
:** SET UP FOR AUTO ANSWER
:CN,18,32B
:** USE CN,18,31B,1B FOR HARDWIRED (NO MODEM)
:** INSTEAD OF CN,18,32B
:** DISABLE PROMPT
:CN,18,21B
::
```

```
FTN4
      PROGRAM XMIT
C  PROGRAM TO TRANSMIT DATA TO THE HP85 VIA 12966 CARD & DVA05
C  LU FOR HP85 (MODEM) IS 18
C
      DIMENSION IDATA(10),IBUF(10)
C
C  SEND SPACE UNDERSCORE (SUPRESSES CR LF) TO CLEAR OUT LINE
      WRITE (18,7)
7     FORMAT(" _")
      WRITE(1,10)
10    FORMAT("SENDING TO HP85....")
C  RECEIVE FIRST HANDSHAKE FROM 85
      IFLAG=2H 0
      CALL EXEC(1,18,IFLAG,1)
      IF(IFLAG .NE. 2HOK)GOTO 999
      WRITE(1,20)
20    FORMAT("ENTER DATA (20 CHARS)--->_")
      READ(1,30)IBUF
30    FORMAT(10A2)
C     WRITE TO HP85
      CALL EXEC(2,18,IBUF,10)
C     SEND THE EOF (##)
      IEOF=2H##
      CALL EXEC(2,18,IEOF,1)
      IFLAG=2H 0
      CALL EXEC(1,18,IFLAG,1)
      WRITE(1,77) IFLAG
77    FORMAT("IFLAG ",A2)
      IF(IFLAG.EQ.2HOK)GOTO 1000
999   WRITE(1,1020)
1020  FORMAT("BAD HANDSHAKE...BYE BYE")
      GOTO 1900
1000  WRITE(1,1200)
1200  FORMAT("DONE...AOK")
1900  END
      END$
```

```
FTN4
      PROGRAM RECV
C GETS DATA FROM HP-85 AND STORES IN DISC FILE ON HP1000
C LU OF HP1000 MODEM IS 18
      DIMENSION JK(40),IDCB(144),NAME(3)
      INTEGER      FLAG
      DATA NAME /2HHP,2H85,2H  /
C     DISC LU WHERE DATA FILE 'HP85' IS LOCATED IS ON LU 40
      ICR=-40
C    OPEN FILE FOR UPDATE MODE--NON-EXCLUSIVE OPEN
      IOPTN=3B
C     SECURITY CODE IS ZERO
      ISC=0
      CALL OPEN(IDCB,IERR,NAME,IOPTN,ISC,ICR)
      IF(IERR .LT. 0)GOTO 999
C
C  CLEAR DATA LINE BY OUTPUTTING SPACE UNDERSCORE (SUPRESS CR LF)
C
      WRITE(18,7)
7     FORMAT(" _")
      WRITE(1,10)
10    FORMAT("HERE COMES THE JUDGE.......")
      FLAG=2H 0
      CALL EXEC(1,18,FLAG,1)
      IF(FLAG .NE. 2HOK)GOTO 101
C CLEAR INPUT BUFFER AND READ
20    DO 21 I=1,40
21    JK(I)=2H
C     READ HP85
      CALL EXEC(1,18,JK,40)
      WRITE(1,40) (JK(K),K=1,40)
C
C   WRITE DATA TO FILE 'HP85' ON DISC
C
      CALL WRITF(IDCB,IERR,JK,40)
      IF (IERR .LT. 0) GOTO 1010
40    FORMAT(40A2)
C     CHECK FOR TERMINATION OF DATA (##) HP85 SENDS '##' TO
C     SIGNIFY END OF TRANSMISSION
      IF (JK(1).NE.2H##)GOTO 20
C WRITE THE DATA FROM HP85 TO SCREEN
      WRITE(1,48)
48    FORMAT("THAT'S ALL FOLKS")
      CALL CLOSE(IDCB,IERR)
      IF(IERR .LT. 0) GOTO 1020
      GOTO 200
```


```
C
C************* ERROR SECTION
101    WRITE(1,102)
102    FORMAT("HANDSHAKE ERROR")
200    GOTO 2000
C
C    FMP ERROR SECTION
C
999    WRITE(1,1000)IERR
1000   FORMAT("OPEN ERROR ",I5)
       GOTO 2000
1010   WRITE(1,1011)IERR
1011   FORMAT("FILE WRITE ERROR ",I5)
       GOTO 2000
1020   WRITE(1,1021)IERR
1021   FORMAT("CLOSE ERROR ",I5)
2000   CONTINUE
C
       END
       END$
```

```
1   ! ANSWER & RECV DATA FROM HP1000 TO HP85
2   ! OVER BELL 212A MODEM AT 1200 BAUD (ASYNC)
3   !   BOB NIEKAMP HP-KANSAS CITY
4   ! SELECT CODE OF HP85 RS232 INTERFACE = 10
5   ! THIS ROUTINE HANDLES ENQ/ACK HANDSHAKE FROM HP1000
6   ! LIKE HP264X TERMINALS.
7   !     PROGRAM  ANS85
10  CLEAR
20  DIM D$[2],P$[45]
30  Q=0 @ R=2 @ B=0
40  ! GO SET UP MODEM BAUD RATE,PARITY AUTO-ANSWER
50  GOSUB 310
55  ! ESTABLISH LINK TO HP1000
60  D$="OK"
70  ENTER 10 USING "#,B";B @ IF B#17 THEN GOTO 70 !INPUT & CHECK FOR DC1
80  OUTPUT 10 USING "K";D$  ! SEND "OK" TO HP1000
90  R=2    !TELLS SUBROUTINE RESPONSE HOW MANY ACK'S TO SEND TO HP1000
100 GOSUB 250  !RESPONSE SUBROUTINE
105 !LINK ESTABLISHED TO 1000...READY TO XMIT DATA
110 R=3 @ DISP "HERE COMES THE CRUD....."  !SEND 3 ACK'S (R=3)
120 !
130 ENTER 10 USING "K";P$
140 PRINT P$  !PRINT DATA FROM HP1000--DATA IS IN P$
150 GOSUB 250  ! RESPONSE
160 IF P$="##" THEN GOTO 180  !CHECK FOR EOT FROM HP1000 PROGRAM
170 GOTO 120
175 ! CHECK FOR DC1 (17)--PENDING READ ON HP85
180 ENTER 10 USING "#,B";B
190 IF B#17 THEN GOTO 180
200 OUTPUT 10 USING "K";D$
210 R=2 @ Q=0  !TELL RESPONSE TO SEND 2 ACK'S
220 GOSUB 250  !RESPONSE
230 DISP "DONE" @ ABORTIO 10
240 STOP
241 !
242 !
245 !*********************************
246 !**  RESONSE SUBROUTINE -- HANDLES ENQ/ACK HANDSHAKE
247 !**
250 ENTER 10 USING "#,B";B
260 IF B#5 THEN GOTO 280
270 OUTPUT 10 USING "#,B";6 @ Q=Q+1  !SEND ACK (6)
280 IF Q<R THEN GOTO 250 !HAVE WE SATISFIED THE 1000 YET??
290 Q=0
300 RETURN
301 !
302 !
303 !***********************
310 !**  MODEM INITIALIZE SUBROUTINE (AUTO ANSWER)
320 !**          GOES IN HERE -- SEE PROGRAM XMT85
330 !  SET BAUD TO 1200
340 !  NO PARITY-- 8-BIT DATA
345 END
350 !
```

```
10 !  ANSWER MODEM & TRANSMIT DATA TO HP1000
20 !  OVER BELL 212A MODEM AT 1200 BAUD ASYNC
30 !     BOB NIEKAMP HP-KANSAS CITY
40 !         PROGRAM XMT85
50 CLEAR
60 ASSIGN #1 TO "BOX1"  !OPEN FILE 'BOX1' ON TAPE TO READ DATA FROM
61 ! NOTE-- 'BOX1' EXISTS & CONTAINS DATA ON TAPE IN HP85
62 !        YOU MUST CHANGE THIS LINE OR CREATE 'BOX1' & PUT DATA IN IT
70 DIM D$[2]          !D$ IS STRING 'OK' TO HANDSHAKE WITH 1000
80 !  SET-UP MODEM
90 GOSUB 550 !MODEM
95 !ESTABLISH LINK TO 1000
100 DISP "WAITING ON 1000..."
110 ENTER 10 USING "#,B";C
120 ! CHECK FOR DC1 (17) FROM 1000---PENDING READ
130 IF C#17 THEN GOTO 110
140 D$="OK"
150 OUTPUT 10 USING "K";D$
155 ! LINK ESTABLISHED
160 GOSUB 300 !DO ENQ/ACK HANDSHAKE
170 DISP "READING DATA "
171 ! READ DATA FROM FILE 'BOX1' ON TAPE
180 READ #1;R$,Z$,S,V @ DISP R$,Z$,S,V @ ASSIGN #1 TO *
190 GOSUB 430 !DO DC1/ACK HANDSHAKE
200 OUTPUT 10 USING "K";R$,Z$,S,V
210 GOSUB 430 !DO DC1/ACK HANDSHAKE
220 OUTPUT 10 USING "K";"##"  !SEND EOT--WE'RE DONE
230 GOSUB 300 !DO ENQ/ACK HANDSHAKE
240 DISP "DATA XMITTED"
250 ABORTIO 10
260 STOP
261 !
262 !
270 !*************************************
280 !** HANDSHAKE SUBROUTINE (ENQ/ACK)
290 !**
300 FOR I=1 TO 2
310 ENTER 10 USING "#,B";B
320 IF B#5 THEN GOTO 310
330 OUTPUT 10 USING "#,B";6
340 NEXT I
350 RETURN
351 !
352 !
353 !
354 !
360 !*************************************
370 !** HANDSHAKE SUBROUTINE (DC1/ACK)
380 !**
390 !**
400 !**
410 !**
420 !**
430 ENTER 10 USING "#,B";T !GET CHARACTER
440 IF T=5 THEN OUTPUT 10 USING "#,B";6 !CHECK FOR ENQ (5)-SEND ACK (6)
450 IF T#17 THEN GOTO 430 !IF NO DC1(17) THEN KEEP HANDSHAKING ENQ/ACK'S
460 RETURN
```

```
470 !
480 !
490 !**********************
500 !**  MODEM INITIALIZE SUBROUTINE (AUTO ANSWER)
510 !**
520 !  SET BAUD TO 1200
530 !  NO PARITY-- 8-BIT DATA
540 !
541 !
542 ! SET DATA RATE SELECT (DRS) TO MODEM FOR AUTO SPEED SENSE
550 ! MUST DO THIS FOR 1200 BAUD
560 ! RS-232 INTERFACE SELCT CODE IS 10
570 CONTROL 10,3;8 !1200 BAUD
580 CONTROL 10,4;3 ! 8-BIT DATA
590 CONTROL 10,2;4 ! DRS SET (DATA RATE SELCT LINE)
600 ASSERT 10;3 @ DISP "WAITING TO BE CALLED..."
610 ! LOOK FOR CTS (CLR TO SEND) ON
620 !WAIT FOR INCOMING CALL
630 STATUS 10,3;M
640 IF NOT BIT(M,1) THEN 630
650 DISP "CALLED"
660 ! START 25 SEC CONNECT TIMER
670 ! ENABLE AUTO-DISCONNECT
680 ! IF DSR,DCD,OR CTS DROP
690 ! THEN DISCONNECT
700 ON TIMER #1,25000 GOTO 860
710 ON INTR 10 GOTO 880
720 ENABLE INTR 10;4
730 CONTROL 10,5;11 ! SET DCD,DSR,CTS
740 ! CHECK DCD
750 STATUS 10,3;M
760 IF NOT BIT(M,3) THEN 750
770 ! IF DCD ON THEN DISABLE TIMER--CARRIER ESTABLIHED
780 ! AND TRANSFER DATA
790 OFF TIMER #1 @ OFF INTR 10
800 DISP "READY"
810 RETURN
820 ! IF TIMER TIMES OUT BEFORE
830 ! DCD ON THEN TURN OFF
840 ! DTR & RTS---SET UP FOR
850 ! NEXT CALL
860 ASSERT 10;0
870 PRINT "ILLEGAL CALL"
880 CONTROL 10,5;0
890 OFF TIMER #1
900 STATUS 10,1;M
910 GOTO 550
920 RETURN
930 END
```

```
10 ! HP85  TO HP1000
20 ! MODEM ORIGINATE MODULE
30 ! USE ALSO FOR HARDWIRE
40 !  BOB NIEKAMP HP-KANSAS CITY
50 !       PROGRAM MOD85
60 ! USE TO SEND DATA TO HP1000
70 ! SETS BELL 212A MODEM FOR 1200 BAUD, NO PARITY
80 ! 8-BITS PER WORD & SET MODEM LINES PROPERLY
90 !    MODEM LINE ABBREVIATIONS:
100 ! DTR= DATA TERMINAL READY
120 ! RTS= REQUEST TO SEND
130 ! DSR= DATA SET READY
140 ! DCD= DATA CARRIER DETECT
150 ! CTS= CLEAR TO SEND
160 ! HP85 RS-232 INTERFACE SELCT CODE=10
170 !
180 CONTROL 10,3;8 ! SET TO 1200 BAUD
190 CONTROL 10,4;3 ! SET 8-BIT DATA WORD
200 ASSERT 10;3    ! RTS,DTR
210 DISP "DIAL NO.--PRESS CONT."
220 PAUSE
230 ! ENABLE AUTO-DISCONNECT
240 ! IF DSR,DCD,OR CTS DROP
250 ! THEN DTR & RTS WILL GO NOT TRUE
260 ! AND GENERATE AN ERROR
270 CONTROL 10,5;11
280 !**********************************
290 !** DATA XMIT ROUTINE GOES HERE
300 !**********************************
```

AVOIDING MAG TAPE LOCKUP

by Bill Hassell/ HP Santa Clara

For some time the number of magtapes on 1000 systems has steadily increased due to their use in data collection and data processing applications. As a result, some of the idiosyncrasies of DVR23 are becoming more noticeable. One of these idiosyncrasies is the system 'lockup' when the 7970E (1600 bpi) tape is rewound and immediately followed by any other motion request. While the reason that the driver locks up the system is buried in both hardware and software, I have been using a very simple workaround that will prevent this lockup condition. A direct benefit is that the routine monitors the break-bit thus allowing graceful termination of a program (if desired) while waiting for the tape to rewind. Basically, the tape rewind is intiated and then periodically tested for BOT (begining of tape or load point) status or the break condition. Since the status bits for CTU's also match DVR23 bits, this routine can be used interchangeably with magtapes or CTU's. Typical usage:

```
      CALL REWWT(LUTAPE)
or
      LOGICAL REWWT
      .
      .   <code>
      .
      IF(REWWT(LUTAPE)) GOTO 900  (Exit on Break)
```

In the first example, the break bit is not needed so the routine returns to the same point whether break was set or BOT was reached. In the second example, REWWT will return .TRUE. if the break bit was set, while REWWT = .FALSE. means that the tape is now rewound and ready for a new command. Since the routine places the program into the time list, it uses no CPU time while waiting for the tape to reach BOT.

A hardware note: the 13183B controller is now available to replace the 13183A controller which causes the lockup condition. This routine is provided to allow older systems to eliminate the problem with a simple software fix, as well as a technique to illustrate the wait EXEC call as a way to avoid wasting CPU time.

```
      LOGICAL FUNCTION REWWT(LUTAPE),REWIND TAPE W/ WAIT REV.2039-BH
C-----------------------------------------------------------------
C
C REWIND A MAGTAPE WITH A 2 SECOND SUSPEND TO ALLOW SWAPPING, AND TO
C   ELIMINATE THE DRIVER 'LOCKUP' FOR PE TAPES.  THIS ROUTINE
C   RETURNS IMMEDIATELY IF THE LU ISN'T TYPE 05, 23 OR 24, OR
C   THE STATUS IS OFFLINE.  DURING THE REWIND WAIT, THE BREAK BIT
C   IS ALSO TESTED WITH THE ROUTINE RETURNING IMMEDIATELY WHEN
C   BREAK IS DETECTED AND SETTING REWWT .TRUE.
C
C SINCE FORTRAN ALLOWS CALLING A FUNCTION AS WELL AS USING ITS RETURN
C   VALUE, USE CALL REWWT(LUTAPE) WHEN YOU DON'T CARE WHETHER BREAK
C   OR BOT WAS DETECTED.   IF YOU NEED TO KNOW THE DIFFERENCE, BE
C   SURE TO DECLARE: 'LOGICAL REWWT' IN EACH PROGRAM UNIT REFRENCING
C   REWWT AND THEN USE SOMETHING LIKE THIS:
C
C                     IF(REWWT(LUTAPE)) ...ACTION...
C
C   THE ACTION IS TAKEN ON BREAK, WHILE FALL THRU TO THE NEXT STATEMENT?
C   OCCURS WHEN TAPE GETS TO BOT.
C
C-----------------------------------------------------------------
      REWWT = .FALSE.
```

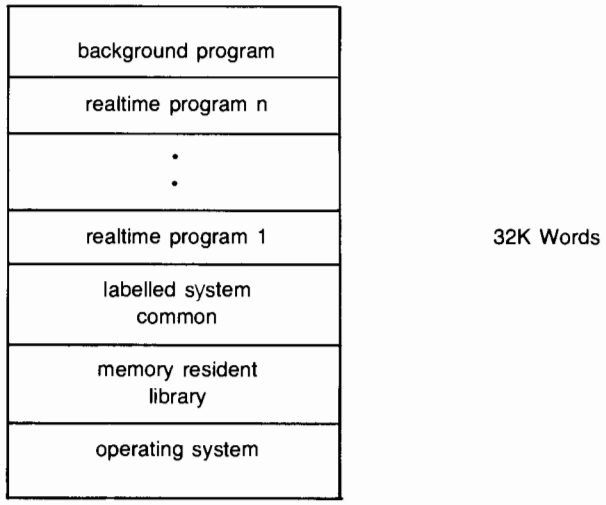
```
C
C USE THE EQT STATUS INFO (DOESN'T ENTER DRIVER) TO BYPASS OBVIOUS
C     ERRORS WITHOUT HAVING TO QUEUE THE REQUEST ON A DOWN UNIT.
C
      CALL EXEC(13,LUTAPE,JEQT4,JEQT5,LUSTAT)

C
C RETURN IF THE UNIT IS NOT AVAILABLE, NOT A TAPE (DVR05/23/24)
C OR ONLINE
C
      JTYPE = IAND(JEQT4/256,77B)
      IF(IAND(LUSTAT,120000B).NE.0.OR.
     +   (JTYPE.NE.05B.AND.JTYPE.NE.23B.AND.JTYPE.NE.24B)) RETURN
C
C REWIND (USE EXEC RATHER THAN REWIND TO ELIMINATE .TAPE MODULES)
C
      CALL EXEC(3,LUTAPE+400B)
C
C NOW LOOP AROUND WAITING FOR THE TAPE TO GET TO BOT
C   USE A DYNAMIC STATUS REQUEST SO THE DRIVER WILL UPDATE WHAT'S
C   GOING ON  (GOES ALL THE WAY THRU THE DRIVER SO THE TABLES
C   ARE UPDATED).
C
10    CALL EXEC(3,LUTAPE+600B)
      CALL ABREG(LUSTAT,IB)
      IF(IAND(LUSTAT,101B).NE.0) RETURN
C
C NOW WAIT 2 SECONDS AND TEST THE BREAK BIT.
C
      CALL EXEC(12,0,2,0,-2)
      IF(IFBRK(LUSTAT).GE.0) GOTO 10
C
C BREAK WAS SET...RETURN IMMEDIATELY WITH REWWT .TRUE.
C
      REWWT = .TRUE.
      RETURN
      END
```

SHARING THE FORTRAN FORMATTER IN RTE-L

by Kent Ferson/HP Data Systems Division

The Fortran formatter is a group of subroutines that allow the user to do I/O or data conversion according to a specified format statement. Any program using a Fortran READ or WRITE statement causes all of the formatter's subroutines (about 3K of code) to be loaded into the user's partition. This article presents a method for sharing the formatter subroutines among several user programs in an RTE-L operating system environment. The mechanics of generating the system are laid out in detail followed by a mathematical analysis of the system's performance.

The RTE-L system has a total of 32K words of memory. Any application with two or more real time (i.e. memory resident) programs can not afford to have independent copies of the formatter for each program. Therefore, it becomes necessary to devise some technique for sharing this code. Examining the memory map of RTE-L, there are three locations where the formatter could reside and still be shared among several programs; the memory resident library, labelled system common, and within a real time or background program's partition. The method presented here uses labelled system common.



RTE-L MEMORY MAP

GENERATION PROCEDURE

The mechanics of relocating the Fortran formatter into labelled system common are a little tricky. The RTE-L generator will only relocate type 14 or type 30 modules into labelled system common. Therefore, the first thing to do is to change the type of the formatter's modules FMTIO, FMT.E, and FRMTR from type 6 and 7 to type 30. The program "TYPE" (see appended code) will do the job. Next, make sure that all JSB's to .ZRNT and .ZPRV are replaced with an RSS opcode. The entry points .ZRNT and .ZPRV have a special meaning to the RTE-L generator. If a JSB to .ZRNT or .ZPRV is encountered while relocating a subroutine into the memory resident library or labelled system common, the generator replaces the JSB instruction with some privileged code. On the other hand, if the JSB to .ZRNT or .ZPRV is encountered while relocating a subroutine into the user's partition, the

generator replaces the JSB instruction with an RSS opcode. Because we want the the formatter's subroutines to logically be part of a user partition, we must force the generator to replace the JSB's to .ZRNT and .ZPRV with an RSS opcode. To do this, relocate the assembly routine "RP" (see appended code) during the generation. The generation answer file should look like this:

```
*
* Relocate labelled system common
*
REL,%RP
REL,$MLIBX,FMTIO,        *  $MLIBX is the output
REL,$MLIBX,FMT.E,        *  of program "TYPE"
REL,$MLIBX,FRMTR,
SE,$SYSLB
SE,$MXLB
SE,$MLIB2
END
```

In order to gain access to labelled system common, each real time or background program must use the "SCOM" command during relocation.

```
*
* Relocate real time programs
*
SCOM
REL,%RTIM1
NEXT
SCOM
REL,%RTIM2
NEXT
 .
 .
END
```

The user programs MUST be written to control access to the shared formatter. This can be done with an RTE-L semaphore called a resource number. The user code should look like this:

```
C
C  Lock the formatter
C
       CALL RNRQ(CNTWD,RN,STAT)
            .
   (perform formatted I/O or data conversions)
            .
C
C  Unlock the formatter
C
       CALL RNRQ(CNTWD,RN,STAT)
```

The resource number must be common to all user programs. See the RTE-L Programmer's Reference Manual for a full discussion on the use of resource numbers.

SYSTEM ANALYSIS

Any scheme that shares a resource among several programs is going to have an impact on the system's performance. We can use a discrete state, continuous transition Markov model (Reference 1 and 2) to analyze the system mathematically. The model is practical because it shows the effect of adjusting system parameters. S(i) represents a particular state where "i" programs need access to the formatter (see figure 1). For example, in state S(3), one program would have access to the formatter and two programs would be suspended waiting for the resource number lock. The probability that the currently executing program will need access to the formatter in the next interval of time "dt" is given by "Wdt". "Udt" is the probability that a program (currently executing in the formatter) will unlock the formatter in the next interval of time "dt".

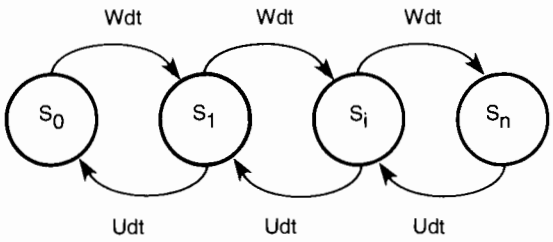


Figure 1

In the steady state, the probability of a S(i) → S(i+1) transition is equal to the probability of a S(i+1) → S(i) transition.

Therefore:

Wdt P(0) = Udt P(1)
Wdt P(1) = Udt P(2)
•
•
Wdt P(n-1) = Udt P(n)

Where P(i) is the probability of being in state S(i). Solving for P(i) in terms of P(0):

P(1) = P(0) (Wdt/Udt)
P(2) = P(1) (Wdt/Udt) = P(0)(W/U)**2
•
•
P(i) = P(0) (W/U)**i

We also know that the state probabilities must sum to one:

Summation i=0 to n P(i) = 1

Using these two facts, we can solve for P(i).

$$P(i) = \frac{(W/U)^{**}i}{\text{Summation } k=0 \text{ to } n \quad (W/U)^{**}k}$$

Observing the geometric series in the denominator, we can easily get a closed form solution.

$$P(i) = \frac{(W/U)^{**}i\ (1-W/U)}{1-(W/U)^{**}(n+1)} \qquad \text{and } W/U <> 1$$

This equation can be used to calculate the individual state probabilites for any given W, U, and n.

Another way to analyze the system's performance is to calculate the expected (or average) number of programs suspended due to lockout at any given instance of time.

By definition:

$$E(\text{suspend}) = \text{summation } i=1 \text{ to } n \frac{(i-1)\ (W/U)^{**}i\ (1-W/U)}{1\ -\ (W/U)^{**}(n+1)}$$

After a little manipulation, the closed form solution is:

$$E(\text{suspend}) = \frac{(n)(W/U)^{**}(n+1)\ -\ (n-1)(W/U)^{**}(n+2)\ -\ (W/U)^{**}2}{(1-(W/U)^{**}(n+1))\ ((W/U)-1)}$$

The model assumes that all the real time programs execute concurrently. This is true for a time sliced dispatching system, but the RTE-L dispatcher always executes the highest priority program. Therefore, the model is only valid for an application where the real time programs are periodically suspended for some reason (i.e. input, output, process synchronization, etc.).

OPTIMIZING PERFORMANCE

There are a number of ways to optimize the system's performance. Looking back at our mathematical model, we see that our best results are obtained when the ratio of W to U is small. One way to do this is to programmatically reduce the time spent in the formatter. If the formatter is used strictly for internal data conversion, we can significantly reduce the time any program will spend in the formatter's code. For example, a typical HP-IB program uses the formatter to convert ASCII data from a device into real numbers. If all the I/O is performed using EXEC calls, the formatter is not locked up while data is being read from the device.

"Inefficient Method"

```
C
C  Lock the formatter
C
      CALL RNRQ(CNTWD,RN,STAT)
C
C  Read data from device
C
      READ(LU,10) R1
 10   FORMAT(F7.2)
C
C  Unlock the formatter
C
```

"Optimized Method"

```
C
C  Read data from device
C
      CALL EXEC(1,LU,BUFF,LEN)
C
C  Now lock the formatter
C
      CALL RNRQ(CNTWD,RN,STAT)
C
C  Convert the ascii to real
C
      CALL CODE
      READ(BUFF,10) R1
 10   FORMAT(F7.2)
C
C  Unlock the formatter
C
```

Another way to reduce the time spent in the formatter is by adjusting system paramters. For example, if the formatter is being used to output data or warning messages to a printer, the buffer limits of the printer could be adjusted to minimize (and maybe eliminate) time spent waiting for the device.

CONCLUDING REMARKS

Earlier I mentioned two other possible locations for the formatter; the memory resident library and within one of the realtime program's partition. Locating the formatter in the memory resident library would work exactly like the labelled system common approach, only the RTE-L operating system would do all the locking and unlocking of the shared resource. The problem is that each subroutine placed in the memory resident library must be coded in a special format, and not all the entry points in the formatter are coded this way. An alternate solution is to have one general purpose program do all the I/O or data conversion. The other programs then use either class I/O, system common, or data files to communicate with the general purpose routine. This is an acceptable solution for an application where all the I/O has a very similiar format, but is not as flexible as the labelled system common solution.

The labelled system common solution works well in a number of applications.

Some examples:

1. HP-IB programs that use the formatter for internal data conversion.
2. Programs that execute sequentially (i.e. the completion of one program triggers the execution of the next).
3. Programs that do their formatted I/O to disc files.
4. Compute bound routines that only occassionally use the formatter.

Any application where several real time programs need access to the formatter will save a great deal of memory space with this technique. Sharing code is especially important in a small system environment where memory space is at a premium. The disadvantage of sharing code is that a high priority program might be suspended when it tries to execute the shareable code.

The same technique could be extended to an RTE-XL operating system environment or to the Fortran 4X formatter with the appropiate modifications (neither one has been tested). Since RTE-XL is a mapped system, the savings would be in physical memory and not logical memory. Please let everyone know if you are successful using this technique with an RTE-XL or FTN4X system.

REFERENCES

1. Drake, A.W.; Fundamentals of Applied Probability Theory, McGraw-Hill Book Company, 1967, chapter 5.
2. Madnick, S.E. and Donovan, J.J.; Operating Systems, McGraw-Hill Book Company, p.240 and p.263.

```
FTN4,L
      PROGRAM TYPE
C
C THIS PROGRAM WAS WRITTEN TO MODIFY 3 MODULES
C TO TYPE 30 IN $MLIB1.  IT HAS VERY LITTLE
C ERROR CHECKING.
C
      IMPLICIT INTEGER(A-Z)
      DIMENSION BUFF(128),DCB1(144),DCB2(144),PBUF(10)
      DIMENSION FNAME(3)
      EQUIVALENCE (PBUF(1),FNAME(1)), (PBUF(5),SC)
      EQUIVALENCE (PBUF(6),CRN), (PBUF(7),TYP), (PBUF(8),SIZE)
      DATA MASK/160000B/, NAM/020000B/
C
C INIT
C
      LU=1
      ILU=128
      ECHO=400B
C
C USER MUST ENTER FULL NAMR (IE. $MLIB1:SC:CRN:TYP:SIZE)
C
      CALL EXEC(2,LU,22HENTER FULL $MLIB1 NAMR,-22)
      LEN=-80
      CALL REIO(1,LU+ECHO,BUFF,LEN)
      CALL ABREG(IA,IB)
C
C PARSE THE FILE NAMR
C
      CHAR=1
      CALL NAMR(PBUF,BUFF,IB,CHAR)
C
C USE FILE NAME TO OPEN $MLIB1
C
      CODE=2HOP
      CALL OPEN(DCB1,IERR,FNAME)
      IF (IERR .LT. 0) GOTO 900
C
C MODIFY FILE NAME TO $MLIBX
C
      FNAME(3)=IAND(FNAME(3),177400B)
      FNAME(3)=IOR(FNAME(3),130B)
      CODE=2HCR
      CALL CREAT(DCB2,IERR,FNAME,SIZE,TYP,SC,CRN)
      IF (IERR .LT. 0) GOTO 900
C
C READ A RECORD FROM $MLIB1 AND CHECK FOR
C A NAM RECORD UNTIL EOF
C
10    CODE=2HRE
      CALL READF(DCB1,IERR,BUFF,ILU,LEN)
      IF (IERR .LT. 0) GOTO 900
      IF (LEN .EQ. -1) GOTO 950
      IF (IAND(BUFF(2),MASK) .NE. NAM) GOTO 100
      CALL MOD(6HFRMTR ,BUFF)
      CALL MOD(6HFMT.E ,BUFF)
      CALL MOD(6HFMTIO ,BUFF)
```

```
C
C WRITE THE BUFFER TO $MLIBX
C
100   CODE=2HWR
      CALL WRITF(DCB2,IERR,BUFF,LEN)
      IF (IERR .LT. 0) GOTO 900
      GOTO 10
C
C COME HERE ON ERROR
C
900   WRITE(LU,910)CODE,IERR
910   FORMAT(" FMP ",A2," ERROR ",I4)
C
C COME HERE TO EXIT
C
950   CALL CLOSE(DCB1,IERR)
      CALL CLOSE(DCB2,IERR)
      END
```

```
FTN4,L
      SUBROUTINE MOD(INAM,IBUFF)
C
C MODIFY NAM RECORD TO TYPE 30 MODULE
C
      DIMENSION INAM(1),IBUFF(1)
      DO 10 I=1,3
      IF (INAM(I) .NE. IBUFF(I+3)) GOTO 100
10    CONTINUE
C
C SAVE OLD TYPE TO UPDATE CHECKSUM
C
      ITEMP=IBUFF(10)
C
C CHANGE TYPE TO 30
C
      IBUFF(10)=30
C
C UPDATE CHECKSUM
C
      ITEMP = 30 - ITEMP
      IBUFF(3) = IBUFF(3) + ITEMP
100   RETURN
      END
```

```
ASMB,L
          NAM RP,30
*
* This module is generated into the system
*
          ENT .ZRNT,.ZPRV
 .ZRNT    RPL 2001B
 .ZPRV    RPL 2001B
          END
```

ACCESSING PHYSICAL MEMORY IN FORTRAN AND PASCAL

By Larry W. Smith/Manufacturing and Consulting Services

In the event that there comes a time in your application where you need to access a portion of physical memory within your program that is not available at load time or otherwise, then this article is for you. In the paragraphs to follow, we will discuss in detail how memory mapping can be accomplished on-line in FORTRAN and PASCAL languages. I will also explain how the user DMS map is constructed and what areas of physical memory they point to.

I could think of no better way to begin the discussion than with an actual example of its usage. There exists an application at the Jet Propulsion Laboratory in Pasadena where security of the individual user and data is of the utmost importance. In order to implement required security procedures, a method had to be devised in RTE-IVB to control the execution of user programs and FMP data files. The contract stipulated that the vendor's standard supplied software must be used without modification for all areas of the application except those areas dealing with security and protection. This left the door wide open for the design of some customized FORTRAN callable routines to implement the required security procedures. The remaining part of this article will discuss the coding of these routines and related topics of DMS.

The entire application of JPL ran under the standard RTE-IVB system with Session Monitor always enabled. Whenever a terminal user ran a program that was not supplied by HP, the program would be allowed to execute long enough to ask for a clearance identification to continue. This is where mapping was utilized. A set of customized assembly routines was designed to initialize, retrieve, and update a table of clearance identification ASCII strings residing in the last two pages of physical memory. I will discuss only one of these routines which performs memory mapping and retrieval. If the clearance identification string entered by the user equals a matching string found in the string table, then execution would continue. Otherwise, the program would update an attempted usage table, issue a message to the terminal user, and terminate. Since the last two pages of memory were left undeclared during system generation and the configurator ($CNFX) was not included, this gave a reasonably sound means of securing access to programs and data files.

In order to see how this can be done, let's assume that the layout of physical memory for the RTE system which was generated for this application is as follows (see page B-9 of "RTE-IVB On-Line Generator Reference Manual"):

The resulting program partition sizes for this 512 kb system were as follows:

```
MAXIMUM PROGRAM SIZE:
W/O COM 29 PAGES
W/  COM 26 PAGES
W/  TA2 19 PAGES
```

The above figures indicate that the largest number of pages that can be mapped into a user's address space when it is dispatched by the system into a partition is 29 for large background (type 4) programs. The application required that most all programs be at least 26k words in length and thus a large background program had to be used. The RTE system would construct the user's 32 DMS mapping registers as follows for a type 4 program without common (refer to figure 1):

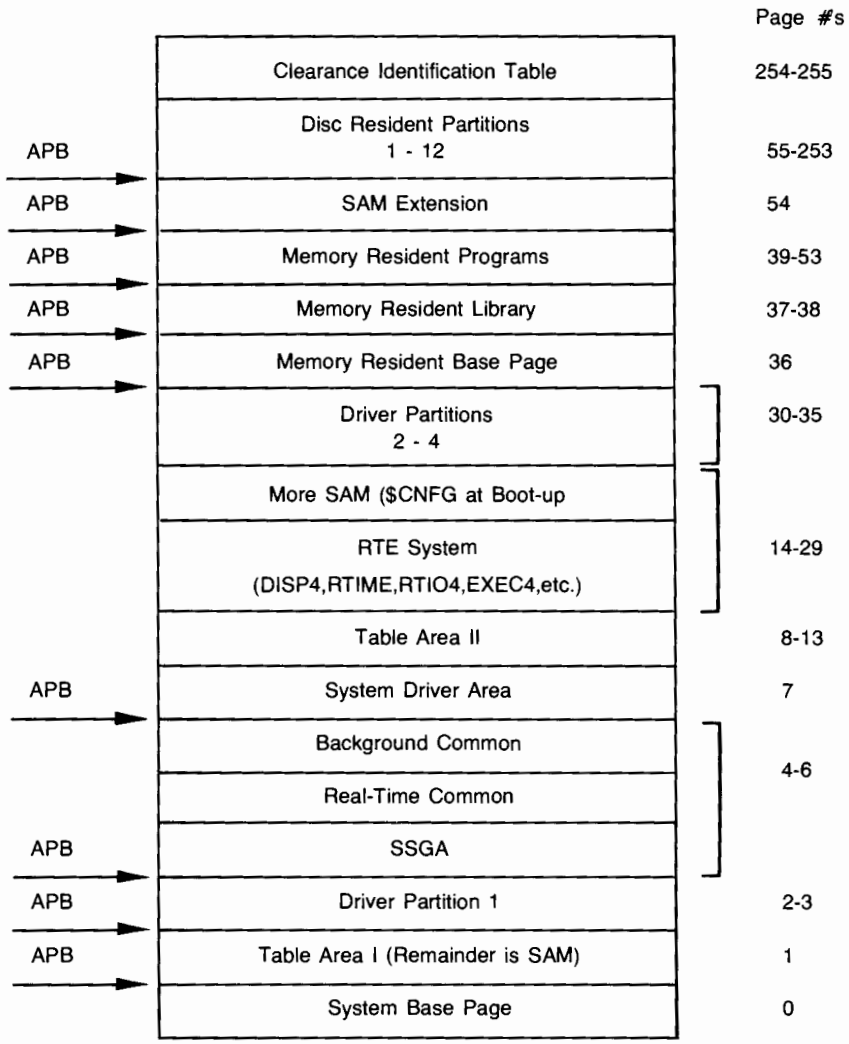
1. Load register 1 with base page from the 4th word in the memory allocation table ($MATA) which describes the characteristics of the chosen partition.

2. Load register 2 with table area I.

3. Load registers 3-4 with driver partition area.

4. Registers 5 through n are loaded sequentially according to the size of the program.

5. The remaining registers are read/write protected.



APB - Aligned on a page boundary.

Figure 1

The end result is that 32 pages of physical memory were selected out of 254 and combined into a 32 k word logical address space with addresses ranging from 0 to 77777 octal. This process is similar to selecting the right options in the purchase of an insurance policy or going to the grocery store and picking what you like best to eat.

To accomplish the actual memory mapping that will allow access to the clearance identification data, a privileged assembly language subroutine called

MAPPER

was written with three entry points that accomplish the following:

MAPON — map in new physical page.

MAPOF — map back old physical page

MEMGT — return contents of mapped physical page.

Once the map is built by the RTE operating system, it is saved in words 2 through 42 octal before the first user loaded module (or local common) only in the event the program is swapped to disc. Thus, when the program executes, it can only be guaranteed to find its map registers by using the appropriate DMS instruction that copies the user map into memory. The USA instruction will allow the contents of the current user's 32 map registers to be copied from the CPU to memory. The routine MAPER illustrates this as follows (altered slightly for this article):

```
ASMB,R,L,Q
       NAM  MAPPER, ACCESS PHYSICAL MEMORY
       ENT  MAPON,MAPOF,MEMGT
       EXT  .ENTR,$LIBR,$LIBX
*
*...MAP IN PHYSICAL PAGE OF MEMORY
*
IPAGE  NOP             PHYSICAL PAGE # TO MAP IN
MAPON  NOP             <<ENTRY/EXIT>>
       JSB  .ENTR      RETRIEVE CALLER'S PARAMETER
       DEF  IPAGE       ADDRESSES.
       LDA  MAPS       GET ADDRESS OF LOCAL TABLE.
       RAL,CCE,ERA
       USA             TRANSFER USER 32 MAP REGISTERS.
       LDB  MAPS+31     SAVE OLD MAP.
       STB  SAVPG      REGISTER CONTAINS.
       LDB  IPAGE,I    GET CALLER'S PAGE NUMBER.
       STB  MAPS+31    AND SET IN LOCAL TABLE.
       JSB  $LIBR      TURN OFF THE INTERRUPT SYSTEM.
       NOP             HERE WE GO!  (ISNT  IT EXCITING.)
       LDA  MAPS
       RAL,CLE,ERA     SET BIT 15=0 TO SET USER MAP.
       USA             PAGE IS NOW MAPPED INTO LAST PAGE.
       JSB  $LIBX      TURN THE LIGHTS BACK ON.
       DEF  *+1        THIS IS THE ONLY GUARANTEED
       DEF  *+1         SAFE RETURN.
       JMP  MAPER,I    RETURN TO CALLER.
 MAPS  DEF *+1
       BSS 32
 SAVPG BSS 1           SAVE OF OLD MAP CONTENTS.
```

(program to be continued in next section of code.)

After the above routine has been called with a valid page number, the user can now use addresses 76000 and 77777 octal to retrieve memory on that physical page. Notice that the user's logical addresses stay the same regardless of what physical page of memory is mapped into it. This is where the term "logical memory" came from. Another way of putting it would be to say that a person stays a person regardless of where he or she might move. The user must ensure that no code or constants are loaded on the last page that is used in conjunction with the above subroutine. For example, if the MAPER subroutine were loaded at addresses 76000 through 76065, then the second USA instruction would have the disastrous effect of mapping itself (MAPER) into oblivion where the next instruction that would be executed would be the eighth word on the newly mapped page.

To retrieve the contents of any location on the newly mapped page, the following routine would be used:

(continued from the previous section)

```
*
*...RETRIEVE LOCATION ON MAPPED PAGE
*
IWORD   NOP             WORD NUMBER
MEMGT   NOP             <<ENTRY/EXIT>>
        JSB .ENTR
        DEF IWORD
        LDA LPAGE       GET START PAGE MEMORY ADDRESS.
        ADA IWORD,I     ADD OFFSET WORD NUMBER.
        LDA 0,I         RETREIVE CONTENTS.
        JMP MEMGT,I     RETURN TO CALLER.
LPAGE   OCT 76000
```

Once a physical page is mapped into last page of the user's address space, then normal access of memory can be accomplished with an indirect load without requiring privileged processing. As an example, let's code a FORTRAN program that retrieves three consecutive words beginning at addresses 7-11 octal on physical page 254:

```
      .
      .
      .
CALL MAPON(254)
IW1=MEMGT( 7b)
IW2=MEMGT(10b)
IW3=MEMGT(11b)
```

In PASCAL/1000, the program would look something like this:

```
program get_physical_memory;
()
 type
  single_integer  = -32768..32767;
  page_number     = single_integer;
  word_offset     = single_integer;
()
var
  iw1 : single_integer;
  iw2 : single_integer;
  iw3 : single_integer;
()
procedure mapon(page_number : single_integer);          external;
procedure mapof;                                        external;
function  memgt(word_offset : single_integer) : integer; external;
()
begin
 iw1 := memgt(7);
 iw2 := memgt(8);
 iw3 := memgt(9);
 END;
 END
```

Finally, the third part of the MAPER subroutine allows the original map to be restored to its former state. The coding would be as follows:

(continued from previous section.)

```
*
*...RESTORE MAPPED PAGE
*
MAPOF   NOP             ENTRY/EXIT
        JSB .ENTR
        DEF MAPOF
        LDA SAVPG       GET OLD REGISTER CONTENTS
        STA MAPS+31     AND PUT BACK IN LOCAL BUFFER.
        LDA MAPS        LOAD BUFFER ADDRESS.
        RAL,CLE,ERA     SET FOR LOAD INTO MAPS.
        JSB $LIBR       SEDATE RTE.
        NOP             CLICK!
        USA             MAP BACK ORIGINAL PAGE.
        JSB $LIBX       WAKE-UP.
        DEF *+1         RTE.
        DEF *+1
        JMP MAPOF,I     RETURN TO CALLER.
        END
```

in order to see how the mapping is accomplished, the following two diagrams show what is mapped before and after the call to MAPON:

Figure 2 shows a typical case where a program is dispatched into a partition beginning at page 149 and ending at page 178. After the call to MAPON is issued, physical pages 254-255 are mapped into the user program area where there were previously two unused and protected pages of memory, namely pages 177-178.

This article has attempted to present a technique of mapping physical memory within a FORTRAN or PASCAL program and also reviews how RTE performs program mapping.

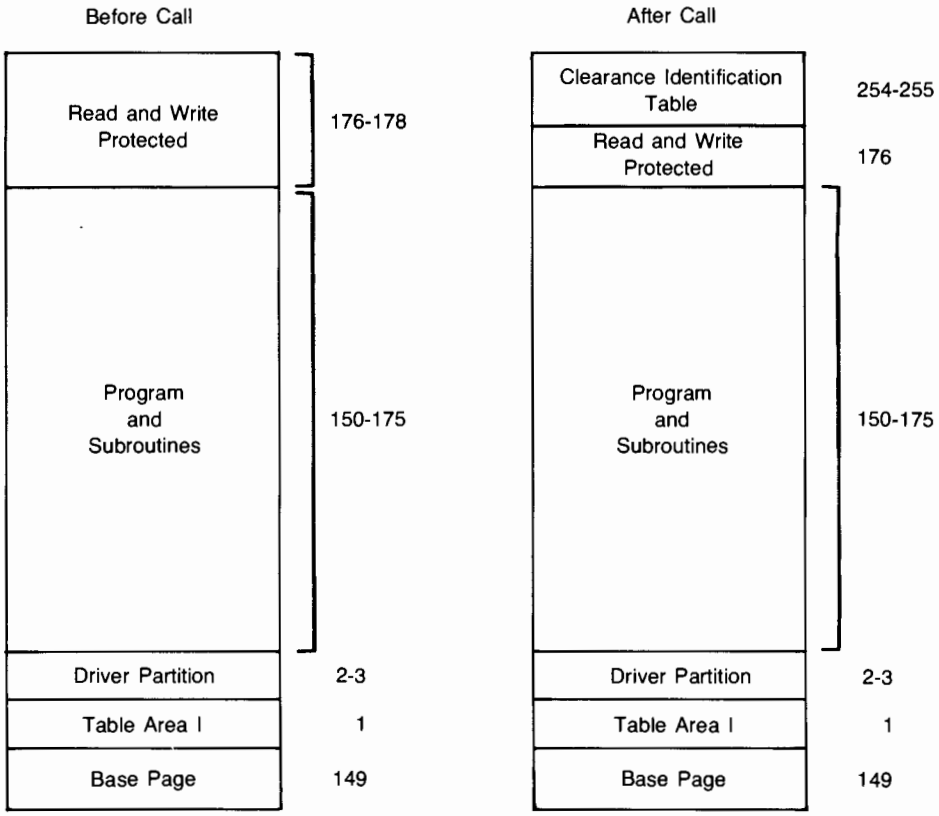


Figure 2

DESIGNING A HIGH-PERFORMANCE DATA-CAPTURE SYSTEM

By Carl Reynolds/HP Rochester, NY

During the past year, I worked closely with a customer as the customer installed a large-scale data-capture system. I think that together we encountered almost all of the difficulties one could encounter in setting up such a system. As the recipient of a summa cum laude degree from the School of Hard Knocks, I now presume to instruct the unwary.

The request for proposal specified 1500 transactions per day. Even though Datacap/1000 performance specifications were yet months away (this was summer, 1979), I had to believe this transaction load would be a trivial burden to the HP1000. The system would serve about 1000 union employees working three shifts at a large manufacturing company. Two functions were to be provided by the system: time and attendance, and job labor accounting. The captured data was to be sent over a DS/1000-3000 link to the HP3000 which would be used for the corporation's other data processing needs.

The hardware selected included an HP3000 with umpty-ump database inquiry/games CRTs, an F-series HP1000 with a megabyte of high-speed fault-correcting memory, 14 3077A time and attendance (T&A) terminals, and 29 3076A data-capture (DC) terminals with type V badge readers and multifunction readers. Of course, both systems were ordered with plenty of disc space, magnetic tapes, printers, etc. The software on the HP1000 included Datacap/1000 and Image/1000.

Excedrin headache number 1 came the day we learned that Datacap/1000 did not have the performance to do time and attendance for first shift. Six hundred workers were to come through the gates in about ten minutes. Datacap/1000 would have needed thirty to fifty minutes to do the job. How would you explain the lines at the gates to an hourly union employee?

I could go on and on about headaches number 2 to 77B. Remember, we were working with what is now called Datacap/1000-I. We used a modified version of TMATT (see Communicator Vol. III, No. 6) to get the time and attendance performance we needed, but then we ran into Datacap/1000-I's exclusive-open problems with databases and files. The most difficult problem of all was getting performance at the DC terminals up to acceptable levels.

A transaction in this application required the selection of one of three similar transactions, the reading and validation (against a database) of a badge, the reading and validation of a punched job number, and the writing of the data to disc and magnetic tape. We did FORTRAN gymnastics to unburden the general-purpose Datacap/1000 software, but the best we could achieve was about 900-1000 transactions per hour. If you look at the performance curves for standard Datacap/1000 (see Communicator Vol. III, No. 5), you will see that we had done relatively well. However, that which impressed us computer jockies was lost on the welders. In this case, 900-1000 transactions per hour meant typical response times of five seconds. With eight active DC terminals and eight people working as fast as possible, response times varied from 5 to 15 seconds.

"So what?", you may be asking. "You only needed 1500 transactions per day."

Right, except that 600 of the 1500 had to occur between 8:00 and 8:10AM. As soon as the employees walked through the gates and identified themselves to TMATT, they were to log-on to a job. TMATT had allowed us to delay total catastrophy by the length of time it took the employees to get to their work stations from the gates. We needed a peak transaction rate of about 4000 per hour.

Well, we solved the problem. The system runs so fast now we really can't measure its performance very easily. With eight or nine people going as fast as they can, response time is virtually instantaneous. We have clocked rates of 6000 transactions per hour, and I am sure higher rates would be observed if we could find people to experimentally work at all 29 stations at once.

I could tell you in detail what we did to achieve this performance, but I think this article will make a greater contribution to the work of others if I instead explain in general what we would do next time. Even better performance should be possible.

Figure 1 shows a block diagram of the system I would design next time. At the center of the system is a validation program. VALID would handle all validations against database information. However, instead of simply maintaining the database on disc, the relevant contents of the database would be loaded into an EMA array. The array subscripts would be determined by hashing the key values, much as Image hashes values to determine record addresses. A means for handling synonyms would, of course, be necessary. The reason for putting the database into memory is speed. A keyed read of a master dataset requires approximately 17ms. Database updates require more time. If the system has enough memory to accomodate the necessary elements of the database, a significant performance gain may be realized by putting the database into memory. The customer was successful in coding such a program for the application with which I worked.

I would have separate programs to handle the T&A and DC terminals. Such modularity would facilitate development, debugging, and operation with no penalty in performance (assuming the system had enough memory to avoid any swapping during execution). TMATT is a very satisfactory program for monitoring the T&A terminals, and it is readily modifiable for application-specific needs. For monitoring the DC terminals, we would have a choice. We could write our own MTM-like class I/O terminal handler, or we could use Datacap/1000's Terminal Management System (TMS) subroutine calls within a Datacap/1000 context. This choice will be discussed in detail later in this paper. Both TMATT and DCTRM would communicate with VALID using class I/O. VALID would be able to distinguish between TMATT and DCTRM requests by looking at the value of one of the optional parameters in the class I/O request.

A separate program would be responsible for storing the data to disc and logging the transactions to magnetic tape. TMATT and DCTRM would use class I/O to pass their data to STORE. This method would ensure that TMATT and DCTRM never had to wait for I/O to the disc or tape. During peak periods, transactions could be buffered in SAM to the extent of SAM availability. This idea was not implemented in the application with which I worked, and I think an appreciable improvement in the system's ability to handle peak loads could be achieved using this plan.

In order to provide maximum responsiveness at the DC terminals, I would want a second CPU, probably an HP3000, to accommodate the other manufacturing applications. Program SEND would read the file written by STORE and pass the information to its slave program on the HP3000. A file on the HP1000 would be used to buffer the data between STORE and SEND so that operations could continue at full speed even if the DS/1000-3000 link went down for a period of time. The file would be designed as a fixed-length-record, circular file with the first record indicating 1) the record last written by STORE and 2) the record last read by SEND. Since both STORE and SEND would need to read and write the same file, a resource number would be used to synchronize the activities of the two programs.

Periodically, information in the HP1000's database and VALID's EMA array would need to be updated (e.g., when an employee was hired or fired). UPDAT would be a slave to a master program on the HP3000, and upon request from the HP3000, UPDAT would 1) communicate with VALID using class I/O (again, using one of the optional parameters in the class I/O call to identify itself as the source of the request), and 2) modify the reference database on the HP1000.

Program CNTRL would provide the means for system start-up and shut-down. When CNTRL was scheduled, it would allocate all the required class numbers and resource numbers and put them where all programs could obtain them. A convenient and secure method for doing this would be to generate into the system a reserved block of memory locations in SSGA. This would require only a simple assembly language program of type 30 which had a BSS statement reserving the necessary room. This module would be relocated with all the other modules at generation time. CNTRL could then put the allocated numbers into this reserved space, and the other system programs could read them out. (Short assembly language subroutines would be required to store the numbers and read them out.) Alternatives would include the use of system common (but Datacap/1000 has its own plans for background system common) or a disc file (but FMP routines appended to our programs would increase the programs' sizes). CNTRL would, in any case, report the allocated numbers on a list device so that in the event of a program malfunction some clean-up of system resources (class buffers, class numbers, resource numbers) might be attempted without recourse to a re-boot of the system.

At start-up, CNTRL would schedule VALID. VALID would load its EMA array from the database and then schedule TMATT, DCTRM, STORE, and SEND. SEND would put itself in the time list for an appropriate time interval so that it would not waste CPU time constantly checking the data file when there was nothing new to transmit.

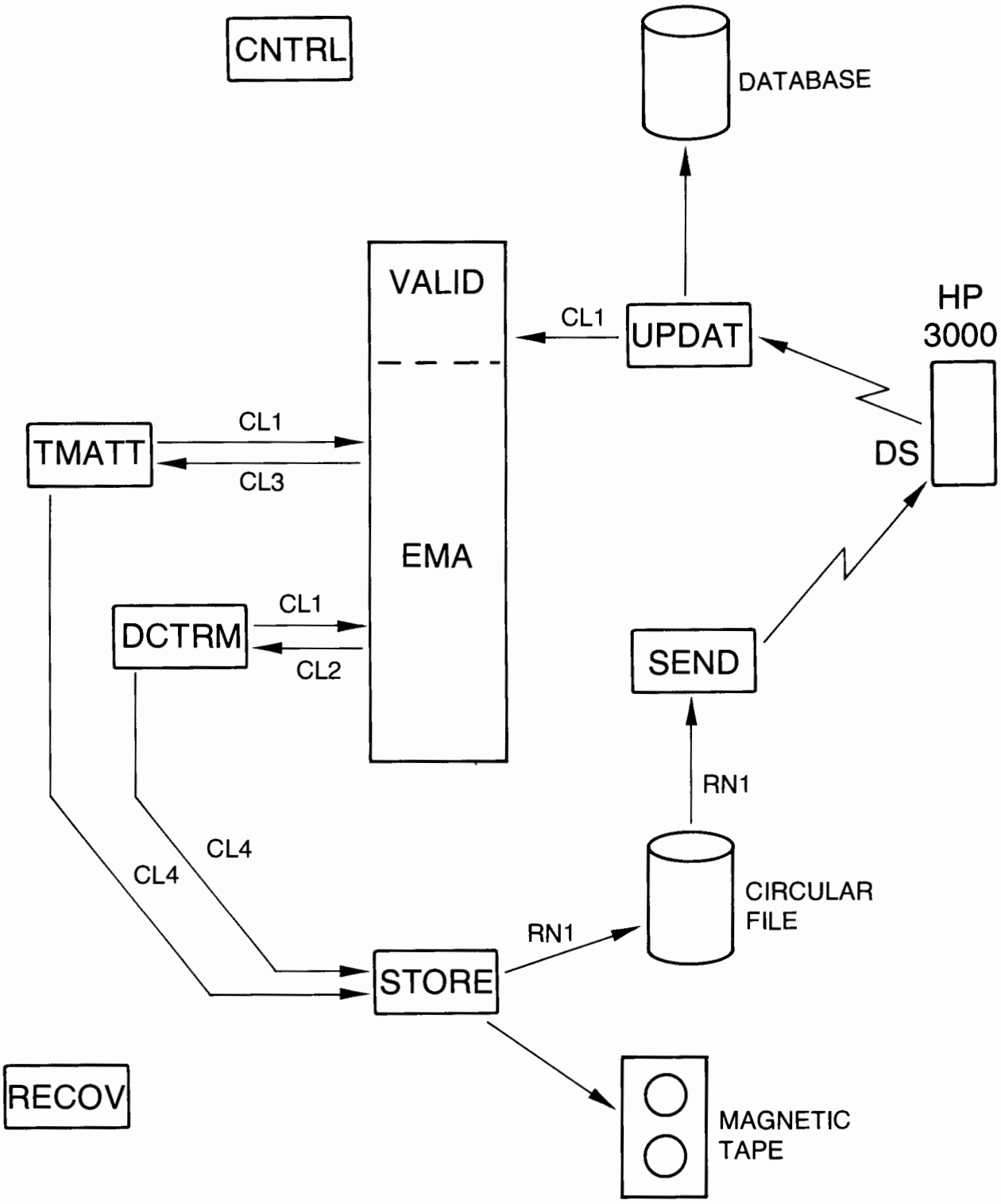


Figure 1. System Block Diagram

CNTRL would have the ability also to perform hard and soft shut-downs. A soft shut-down request would cause CNTRL to set the break bits for VALID and STORE (via a call to MESSS). VALID would stop honoring new validation requests (and pass back a "shut-down" flag in one of the optional parameters of the class I/O call), and TMATT and DCTRM would know then to 1) deallocate their particular class numbers, 2) "sign-off" to VALID using VALID's class number, and 3) terminate. STORE would 1) honor pending storage requests, 2) close the disc file, 3) deallocate its class number, 4) set the break bit for SEND, and 5) terminate. SEND would 1) transmit the remaining data records to the HP3000, 2) deallocate the resource number, 3) close the data file, 4) close PTOP with its slave, and 5) terminate taking itself out of the time list. When VALID received the "sign-off" messages from both TMATT and DCTRM, it would 1) deallocate its class number, and 2) terminate.

A hard shut-down request would cause CNTRL to abort VALID, TMATT, DCTRM, STORE and SEND, and itself deallocate all class numbers and resource numbers. A hard shut-down might be necessary when the system was in place but inactive. Most of the programs would be waiting on class I/O GET requests, and they would therefore be unable to examine the states of their break bits.

One last module, called RECOV, would allow the reconstruction of most transactions in the event of some kind of system failure. RECOV would read the logging tape and retransmit specified transactions to the HP3000.

Having described the data-capture system as a whole, I would like to discuss the choice of DC terminal handling routines. Whether we wrote our own class I/O terminal handler or worked within a Datacap/1000 context, we would have to become familiar with the escape sequences which control the terminals. The Quick Reference Guide for the 3075/6/7A terminals (part no. 03075-90002) is probably all we would need. Whichever choice we made, we would be writing our own code to display messages, turn on prompting lights, read data from the keyboard, enable badge readers, etc. Controlling the terminals ourselves, rather than letting Datacap/1000's general-purpose routines control them, would make, by far, the greatest contribution to system performance of all. The other tricks, such as a database in memory or SAM data "storage", could improve performance by a factor of two, but controlling the terminals ourselves could improve performance by an order of magnitude or more.

Our programming job would be easier if we worked within a Datacap/1000 context. Datacap/1000 uses a Terminal Management System (TMS) to achieve a kind of reentrancy (called "breakpoint reentrancy") for routines under its control. Code and data are separated by putting all data in background system common, and TMS keeps track of the terminal context (i.e., which terminal the code is now addressing) for us. TMS manages system common so that whenever a TMS I/O operation completes, the present contents of system common are saved in an EMA array, and the data for the particular terminal needing service are written back into system common. Thus we could write our subroutines as though we were concerned with only one terminal when in fact we were controlling forty. The DATA, VALIDATION, DISPLAY, and STORAGE routines documented in the Datacap/1000 manual (part no. 92080-90001) are good examples. Once one of our Datacap/1000 transactions called one of our user subroutines, we could use the TMS subroutine calls directly, as described in the "Advanced Techniques" chapter of the manual, to write to and read from the terminals.

Specifically, to make use of TMS to control the DC terminals, we would employ Datacap/1000 as follows. Using TGP, we would generate a "dummy" transaction. This dummy transaction would specify only one U-type question, and it would specify that a user-written display routine was to be used to generate a character string display of, say, two characters in length. The dummy transaction would also specify the automatic complete/select capability, and the "captured data" would be stored to some dummy disc file.

The whole point of the dummy transaction would be to invoke the user-written display subroutine. In actual practice, control would never return from our user subroutine; our "display" subroutine would use the TMS calls to write escape sequences to the terminals, read data from the terminals, validate the data, and store the data as required. If it were ever appropriate to return control to Datacap itself, our subroutine would simply generate a two-character string and return. Thanks to our display and data subroutines, a four-character dummy record would be written to the dummy file, and the terminal would return to the select-transaction state.

From within our display subroutine, we could make use of any of the TMS subroutines. There are six TMS routines described in the Datacap Manual. Of these routines, four are most likely to be useful to us: TMDFN, TMRD, TMBWR, and TMPZ.

The first executable statement in a TMS routine must be a call to TMDFN. TMDFN defines up to six groupings of reentrant variables (called common blocks) in system common. The groupings are defined by passing to TMDFN first-word and last-word-plus-one addresses for various common blocks (CBs). Actually, CBs must not only be defined but also "enabled". Ordinarily, the calling routine enables the subroutine's CBs by virtue of the fact that enabled CBs in the caller are also enabled in the subroutine, if the CBs are defined in the subroutine. Only those variables stored in the defined and enabled CBs will be saved whenever a TMS service request (i.e., a call to a TMxxx routine) is issued. This is very important to remember; any data that must be preserved over one or more TMxxx calls must be stored in common! An error here will only show up when several terminals are active; you can think your programs are completely debugged until you call your manager in, bring the system on-line, and discover right in front of the whole world that all your data is GARBAGE! (Yes, I did.)

The form of the TMDFN call is as follows:

```
CALL TMDFN(ICB0,ICB1,ICB2 [,ICB3 [,ICB4 [,ICB5 [,ICB5E]]]]
```

where,

ICB0 = 1st word of CB0
ICB1 = 1st word of CB1 or last-word-plus-one of CB0
ICB2 = 1st word of CB2 or last-word-plus-one of CB1
ICB3 = 1st word of CB3 or last-word-plus-one of CB2
ICB4 = 1st word of CB4 or last-word-plus-one of CB3
ICB5 = 1st word of CB5 or last-word-plus-one of CB4
ICB5E = last-word-plus-one of CB5

A common block may be omitted by either leaving it off the end of the list or specifying that its starting and ending addresses are the same.

For example, look at the TMDFN call for the user-written display subroutine. The manual tells us to write the following code:

```
      SUBROUTINE DISPL
      LOGICAL BKSFL
      COMMON KEEP1(3),ISTAT,ITLOG
      COMMON KEEP2(5),LUQ,LMQ,IBUF(512)
      COMMON ITSNU,INDEX,IQNUM,ITMTP,ITMLN,IBUPT,BKSFL,INBKS,IQBKS
      COMMON IER,NSTAT,IUTRN,IUINP(64),ITEMP(256)
      COMMON KEEP3(210),IMGBUF(512)
      COMMON ICOMEN
C
      CALL TMDFN(KEEP1,KEEP2,ITSNU,ITSNU,KEEP3,ICOMEN)
```

The TMDFN call defines CB1, CB2, CB4, and CB5. CB0 and CB3 are not defined and not used. CB1 has information about the particular terminal the routine is addressing at any moment. KEEP1(1) contains the LU, and ISTAT and ITLOG contain the status and transmission log. CB2 has the data collected thus far in the transaction with this terminal. CB4 has a number of pointers and flags as well as the buffer for data subroutine use and the buffer for temporary storage of reentrant variables. CB5 is used when the transaction has access to a database.

So, what is there to know about using TMDFN? Usually, we have only to follow the cookbook in the manual precisely. However, in our case we could reduce system overhead by not defining CB5 (omitting the fifth COMMON statement and the sixth subroutine parameter in the code above), if we were not going to access a database in our transaction.

TMBWR and TMRD would be used heavily in any code we would write to control the terminals. TMBWR is a buffered (i.e., no wait) write call, and TMRD is a read call. The calling sequences are well described in the "Advanced Techniques" chapter of the manual. As an example, the following code would disable the keyboard and the type V badge-reader, enable the multifunction reader (holes not marks, no clock, ASCII not binary, corner cut detection enabled, "single field" operation not buffering within the terminal), turn off all prompting lights, turn on prompting light 2, clear the display, write "INSERT CARD", and read all 80 columns of a punched card. For those interested in following the escape sequences, Escape- equals 015455B, and EscapeJ equals 015512B.

```
      DIMENSION DBKEM(22)
C
      DATA DBKEM /015455B,2Hc0,2Hb0,2Hk1,2HR  ,
     &            015455B,2Hr0,2Hn0,2Hi1,2Hc0,2HM  ,
     &            015455B,2Hd0,2H(1,2HA  ,
     &            015512B,2HIN,2HSE,2HRT,2H C,2HAR,2HD /
C
  300 CALL TMBWR(DBKEM,22)
   70 CALL TMRD(ITEMP(5),40)
```

TMPZ has several uses. It may be called either with or without a time parameter (see the Datacap manual) expressed in tens of milliseconds. TMPZ permits a pause in the processing for the terminal currently being addressed, and it allows TMS to handle other terminals while processing has paused for the one. When TMPZ is called with no parameter, all other pending processes (i.e., requests from other terminals) are serviced before TMS returns to the paused process. When a time parameter is specified, the process is paused, and other processes serviced, only as long as the time specified.

One occasion on which to use TMPZ is when we know that our subroutine would have to wait anyway. For instance, the subroutine might do a class I/O WRITE/READ to VALID to check a badge, and then do a class GET to wait for and receive the reply. We would want to call TMPZ between the WRITE/READ and the GET so that other terminals could be serviced during the time that otherwise would be wasted while the subroutine waited on the GET.

In many cases, it will make sense to call TMPZ with no time parameter. However, there is appreciably less system overhead incurred when a TMPZ call does specify a time, so if we could guess at a reasonable time to specify, we could really optimize our code.

Another occasion on which to use TMPZ is when we want certain timing between messages to the terminal. For instance, in the event of an error, it would often be nice to display an error message long enough for the employee to read it, and then redisplay the question. The following code would display an error message, wait two seconds, and then GO TO 10 to redisplay the question.

```
CALL TMBWR(ERRMS,14)
CALL TMPZ(200)
GO TO 10
```

The TMS calls I have discussed are all the calls we would need to write the sort of terminal handling routine I have in mind. Writing our terminal handler within the TMS framework would certainly simplify the programming task, and it would leave us within the Datacap/1000 system so that we could easily shift back to standard Datacap transactions whenever we chose to and whenever performance requirements permitted.

The alternative terminal control routine would be a program using class I/O to address all the DC terminals at once. We would use class I/O writes (EXEC 18) to send our escape sequences and messages to the individual terminals, and we would use class I/O reads (EXEC 17) to collect data from the terminals. Most students of the RTE Users' Course have had some experience writing an elementary class I/O terminal handler. In addition to concerning ourselves with terminal escape sequences, rules and methods of validation, and plans for data storage, we would also have to think about the following questions. How would the routine learn which terminals were to be addressed? How would the program keep data from each terminal distinct from data from every other terminal? How would the program keep track of the transaction flow for each terminal? How would the program put an error message on a terminal for two seconds, and then ask the question again, without suspending the entire operation for that period of time?

These challenges could certainly be met, but I am sure you will agree that these challenges would be far greater than those we would face if we stayed within the TMS framework. I am confident that a class I/O terminal handler would substantially outperform a TMS routine doing the same job, but there would be a price to pay in terms of program development, flexibility, and maintenance. If adequate performance could be obtained using TMS, that method would probably be the method of choice. In the application with which I worked, we used TMS routines to achieve our transaction rates of upwards of 6000 per hour.

Datacap/1000 is a terrific tool for implementing a data-capture system, and I hope this paper has suggested ways in which one could expand the capabilities of an HP1000-based data-capture system to include applications which off-the-shelf Datacap/1000 would otherwise be incapable of addressing. There is an enormous reserve of performance in the HP1000 computer, and it can be tapped in several ways in order to meet exceptional demands placed upon the system.

A COMPARISON OF HEWLETT-PACKARD PASCAL/1000 WITH UCSD PASCAL

By John Stafford/HP Data Systems Division

One of the most frequently asked questions about Pascal/1000 is how it compares with other Pascal implementations, most notably UCSD Pascal(1).

UCSD Pascal is a P-code compiler which runs under the UCSD Pascal system. The UCSD Pascal system comprises not only the compiler but also a screen oriented editor, a file system manager, an adaptable assembler, a linker, and a number of other utilities. It is in fact a complete single user operating system. Most of the UCSD Pascal system is written in the UCSD Pascal language. The UCSD Pascal compiler (as noted above) generates P-code, which are instructions for a hypothetical stack-based computer. A P-code interpreter program which runs on the host computer "executes" these P-code instructions. This makes the UCSD system extremely portable, because once a P-code interpreter is written for a new machine, the entire system can be used on that machine. The resultant availability of the UCSD Pascal system (and thus compatible versions of the UCSD Pascal language) on a number of microprocessor-based systems have made it very popular with small computer users.

Pascal/1000 is a native code compiler which runs under the RTE-IVB operating system. The RTE-IVB system provides an editor, file system manager, assembler, linker, and other utilities that are available to the user of Pascal/1000. Pascal/1000 produces assembly code for the 1000 series machines which is automatically assembled to produce a relocatable file. As such, the compiler is not portable to other machines (nor is the RTE-IVB system or other utilities).

This article compares the languages and to some extent the runtime support provided. It does not deal with the other aspects of the UCSD Pascal system or the RTE-IVB system (editor, loader, other utilities) except as required. Hence the term UCSD Pascal will for the purposes of this document refer to the language compiled by UCSD Pascal compiler and the runtime support provided by the UCSD Pascal system.

It is assumed that the reader is familiar with the basics of RTE-IVB, as terms like partition, EMA, and segment will not be explained any more than the context requires. Some knowledge of Pascal/1000 will be useful and access to the Pascal/1000 Reference Manual (92832-90001) is assumed. Any unqualified references to a chapter or appendix refer to that manual. Detailed knowledge of UCSD Pascal is not assumed. See the references for sources of information on UCSD Pascal, Pascal/1000, and Pascal in general.

(1) UCSD Pascal is a trademark of the Regents of the University of California. All further references to UCSD Pascal in this document acknowledge that fact.

SYNTACTIC AND SEMANTIC DIFFERENCES

Program Heading: Pascal/1000 permits file names to be specified in the program heading and will automatically associate them with the external file names specified in the run string at execution time. If the files INPUT and/or OUTPUT appear in the program heading, their appearance represents their declaration, and they must not be declared in the program VAR section (although local files of those names can be declared in procedures or functions). If INPUT appears, it will automatically be RESET when execution starts. If OUTPUT appears, it will automatically be REWRITTEN when execution starts. Files other than INPUT/OUTPUT which appear in the program heading must also be declared in the program VAR section so their type will be known. When any of these other files are opened (with RESET, REWRITE, APPEND, or OPEN) the external file specified in the runstring (associated by position in the list) will be accessed.

UCSD Pascal permits file names to be specified in the program heading but the specification is ignored. The standard files INPUT, OUTPUT, and KEYBOARD are always declared and will be opened when execution starts. Additional files must be declared in the program VAR section.

Segment Procedures: Pascal/1000 does not provide a mechanism which corresponds exactly to this feature. A compilation unit may be designated a segment (with the SEGMENT compiler option) and must be explicitly loaded by the users program before any routine in it may be called. Only one segment may be loaded at a time and segments may not load other segments (only the main program can load segments).

UCSD Pascal permits a procedure or function to be specified such that it will only be loaded into memory from an external storage medium when it is actually needed. The load is automatic when the routine is called, if it is not already in memory. Segment routines can call other segment routines subject to the limitation that only six such routines (not including the main program) may be in memory at any given time.

Include Files: Pascal/1000 has a compiler option that permits a file to be included (merged in at compile time) in the source. The included file may not itself contain an include option. The inclusion is a simple textual substitution; the file is literally inserted in the source at the specified place and no other restrictions apply.

UCSD Pascal has a compiler option that permits a file to be included. The inclusion has the same properties and limitations as that of Pascal/1000 (except see 'Declaration order' below, for an additional special property of UCSD Pascal include files).

Identifiers: Pascal/1000 identifiers must start with a letter and can contain any letter, digit, or the underscore character. They are normally significant for their entire length (150 characters is the maximum line length, hence the maximum identifier length, as an identifier cannot be split across a line). Upper and lower case are considered identical. There is a compiler option to limit the number of significant characters in identifiers to anywhere in the range 1 to 150 characters.

UCSD Pascal identifiers must start with a letter, can contain any letter or digit, and are significant for only 8 characters. They may be longer than 8 characters, but only the first 8 characters are used when distinguishing identifier names. It is not known if lower case letters are accepted at all, and if accepted if they are considered identical to upper case letters.

Declaration Order: Pascal/1000 permits declarations of CONST, TYPE, and VAR to appear in any order and to be repeated as necessary. A LABEL declaration, if present, must come first. CONST, TYPE, and VAR may not appear after any PROCEDURE or FUNCTION declaration.

UCSD Pascal requires CONST, TYPE, and VAR to appear in that order (and after LABEL if present) and each only once. The exception is that if an include file is included after the last VAR declaration and before the first PROCEDURE or FUNCTION it may contain another declaration section (i.e. LABEL, CONST, TYPE, VAR, and PROCEDURES or FUNCTIONS).

GOTO: Pascal/1000 allows a label to only appear in the statement list of the block in which it was declared. It allows a GOTO to branch within the current block or to any outer block. The results of branching into a compound statement are undefined. The environment is not cleaned up if a branch is made out of a recursive invocation of a procedure or function and the results of calling that routine again later are undefined.

UCSD Pascal has a compiler option which allows or disallows the presence of GOTO statements (and LABEL declarations?) in the source. If allowed, a GOTO may only branch within the same block. The results of branching into a compound statement are undefined.

EXTERNAL Routines: Pascal/1000 can call any routine which uses either the .ENTR or DIRECT (no DEF to the return address) calling sequences. Thus it can call PASCAL, FORTRAN, C, assembler, and system library routines which follow these conventions (PASCAL, FORTRAN and C always do). Such external routines are accessed by declaring the procedure heading and replacing the body with the directive EXTERNAL. The compiler option DIRECT must be specified if the routine uses the direct calling sequence, and the compiler option ALIAS may have to be used if the routine has a name which is not a valid Pascal identifier. Calls to the routine are checked by the compiler just as if the routine were present; however no check is made for number, type, or number of words of arguments being passed when the program is loaded.

UCSD Pascal can call UCSD Pascal, UCSD Fortran 77, and host machine assembly language routines by making an equivalent EXTERNAL declaration. The rules for Fortran and assembly language calls may be found in UCSD Pascal system documentation. The total number of words of arguments (but nothing else) is checked for correctness by the loader.

CASE Statements: Pascal/1000 allows the specification of a range of values for a case label, e.g. (assuming 'i' to be INTEGER or subrange)

```
CASE i OF
    1:    ...
    2..10,20:   ...
    99:   ...
    OTHERWISE
        ...
END;
```

It also permits the presence of an OTHERWISE clause to which control will be transferred if none of the explicit cases is matched. A runtime error will occur if none of the explicit cases is matched and no OTHERWISE clause is specified.

UCSD Pascal does not permit the specification of a range of values nor does it have an OTHERWISE clause. If none of the explicit cases is matched, control will simply fall through to the end of the case statement. This falling through behavior can be performed in Pascal/1000 by having an OTHERWISE clause containing no executable statements.

Comments and Compiler Options: Pascal/1000 permits both kinds of comment delimiters {...} and (*...*) to be used. A comment must end with the matching delimiter. This permits code which contains comments to be commented out with the other type of comment delimiters.

```
{the following code is commented out
FOR i := 1 TO 10 DO BEGIN (* process the data *)
    a[i] := a[i] * [i] + c[i * 2];
    IF a[i] < 0 THEN BEGIN
       a[i] := 0; (* negative treated like zero *)
    END;
END;
down to here}
```

Comments of the same type do not nest, so the following will cause a syntax error somewhere in the 'not here *)' part.

```
(* this is a comment (* but it ends here *) not here *)
```

UCSD Pascal has the same rules for comments but with the additional feature that if the first character after the start of comment delimiter is $ then the rest of the comment is a compiler option. UCSD has a number of compiler options, (B,C,D,G,I,L,P,Q,R,S,U) which control things like: whether or not to generate a listing and to what file, whether GOTO should be allowed, whether range checking should be done, where to eject a page in the listing, where to include a file, and several others. UCSD documentation should be consulted for more information.

Pascal/1000 does not recognize compiler options within comments. They are specified outside of comments but within a pair of $'s. If the trailing $ is not present in a compiler option, the option is assumed to stop at the end of the source line. Pascal/1000 has a number of compiler options that do things similar to those done by the UCSD compiler (except GOTO's are always allowed) as well as things that are unique to Pascal/1000 (see Appendix D).

Procedures and Functions as Formal Parameters: Pascal/1000 allows both procedures and functions as formal parameters. The parameter list must be specified, so it can likewise contain a procedure or function formal parameter. The parameter lists of actual parameters must match (Chapter 4) those of the formal parameters. Standard procedures and functions may not be actual parameters.

UCSD does not permit either procedures or functions as formal parameters. The UCSD Pascal intrinsic EXIT does take either a procedure or function identifier as its argument.

Extended Comparisons: Pascal/1000 permits the operations =, <>, <, <=, >=, and >on arrays of element type CHAR, either PACKED or not PACKED. No comparisons are permitted on any other type of array or on any type of record.

UCSD Pascal permits the operations = and <> between objects of any array or record type.

Units: Pascal/1000 has no feature which is significantly similar to the UCSD Pascal concept of a UNIT. Pascal/1000 does permit the separate compilation of groups of level 1 procedures and functions either to be relocated with a program or segment (the SUBPROGRAM option), or to be a dynamically loadable segment (the SEGMENT option). Both of these objects must use the same globals as the main program (if they use any globals at all) and they do not have their own private data area (they may have their own type definitions, and the routines can have their own local data).

UCSD Pascal has a separate compilation facility called a UNIT. The following paragraph, quoted from the UCSD Pascal System II.0 User's Manual, is a good description of a UNIT.

"A UNIT is a group of interdependent procedures, functions, and associated data structures which perform a specialized task. Whenever this task is needed within a program, the program indicates that it USES the UNIT. A UNIT consists of two parts, the INTERFACE part, which declares constants, types, variables, procedures, and functions that are public and can be used by the hostprogram, and the IMPLEMENTATION part, which declares constants, types, variables, procedures, and functions that are private. These are not available to the host program and are used by the UNIT. The INTERFACE part declares how the program will communicate with the UNIT while the IMPLEMENTATION part defines how the UNIT will accomplish its task."

A UNIT cannot access the globals of the host program, but the host can access the INTERFACE variables of the UNIT. If a host program uses a unit a USES <unitname> declaration must come after the program heading and before the first declaration. Name conflicts can occur if a host program global identifier is the same as an INTERFACE identifier in the UNIT. A procedure or function may not USE a UNIT locally. The Turtle Graphics facility of UCSD Pascal is an example of a UNIT.

TYPE DIFFERENCES

Integers: The predefined type INTEGER in Pascal/1000 is a two word integer with a range of -2,147,483,648..2,147,463,647. A one word integer may be declared as a subrange -32,768..32,767 (or any subrange within that range inclusive). No larger integers are provided.

The predefined type INTEGER in UCSD Pascal is a one word integer with a range of -32,768..32,767. Larger integers are declared as having a type INTEGERn (1 <= n <= 36) where 'n' is the number of digits. The Pascal/1000 predefined type INTEGER can hold all the values of UCSD Pascal INTEGER1 through INTEGER9 (actually up to INTEGER10 if within the range noted above).

Sets: Pascal/1000 limits sets to a maximum of 32,767 elements. Sets of sixteen or fewer elements fit in one word (or less if a part of a packed structure). Sets of more than 16 elements require (number of elements) DIV 16 + 1 words of storage (whether or not part of a packed structure). The extra word is allocated at the beginning of the set and holds its current length. If a set denotation with integer components appears in an ambiguous context it will be assumed to be a set of 0..255. A set denotation may be explicitly typed by placing a type identifier immediately before it, e.g.

```
TYPE
    BIGSET = SET OF 1..1000;

y := 500;

{this won't do what we want, compiler assumes 0..255}
IF x IN [10, 40, 70, y] THEN

{this will do what we want, compiler knows better}
IF x IN BIGSET [10, 40, 70, y] THEN {it's a BIGSET}
```

UCSD Pascal limits sets to a maximum of 4,080 elements. It is not known what it does with ambiguous sets, but presumably it allocates the largest possible set (which was deemed a potential waste of space in Pascal/1000, 32,767 elements is 2,049 words).

Extended Precision Real Numbers: Pascal/1000 has a predefined type LONGREAL which is similar to type REAL. Its magnitude limits are identical to type REAL but it allows more significant digits of precision. A variable of type LONGREAL requires four words, a variable of type REAL requires two words.

UCSD Pascal does not have the LONGREAL type. A variable of type REAL requires two words. The magnitude limits and number of significant digits is almost the same as that for the Pascal/1000 type REAL.

Arrays: Pascal/1000 does not have a limit on the number of array dimensions or the number of elements in a dimension except that the total number of elements must not be greater than 2,147,483,647. Note that an array larger than 32,767 words could not be declared as a variable, but it could be declared as a type and a dynamic variable of that size might be allocatable if the heap is in the EMA area.

UCSD Pascal has a limit of 16,384 words in any one dimension.

Strings: Pascal/1000 has no predefined string type. An array of characters, either PACKED or not PACKED, is considered a string. The length of the array is user defined, and if only part of the string is valid this length information must be kept by the user. String constants are considered PACKED ARRAYS OF CHAR. Two strings may be compared (see Extended Comparisons). The shorter will be blank filled to the length of the longer and packing/unpacking will occur as needed. A string can be assigned to another string if the source is of the same length or shorter than the destination. Blank filling and packing/unpacking will occur as needed.

UCSD Pascal has a predefined type STRING which is essentially a PACKED ARRAY [1..80] OF CHAR with an associated length value of type 0..255. The default length may be overridden (i.e. STRING[100]) and the maximum string length is 255. String operations and intrinsics are provided (LENGTH, POS, CONCAT, COPY, DELETE, INSERT, and STR). Additional intrinsics for PACKED ARRAY OF CHAR include SCAN, MOVELEFT, MOVERIGHT, and FILLCHAR (it is not clear if these can be used on the STRING type).

PACKED:

```
ARR = PACKED ARRAY [1..10] OF PACKED ARRAY [1..8] OF BOOLEAN;
REC = PACKED RECORD
          sign: BOOLEAN;
          digit: PACKED ARRAY [1..5] OF 0..7;
      END;
```

Pascal/1000 packs data into words from high order bits toward low order bits. If an object cannot fit in the remaining bits of the word currently being packed and it must be word aligned (see below) then some remaining low order bits may be unused.

PACKED in Pascal/1000 may be effective on more than the last occurence of ARRAY in a declaration. Array ARR (above) uses five words. Pascal/1000 packs by the rules that an object requiring a word or less of storage will not cross a word boundary (it will be aligned at the next word boundary if it won't fit in the word currently being packed), and an object requiring more than a word of storage will always be aligned on a word boundary. So in the above example, each element of the PACKED ARRAY [1..8] OF BOOLEAN requires one bit, so each PACKED ARRAY [1..8] OF BOOLEAN requires eight bits, so two such objects can be packed per word.

The fields of PACKED RECORDS in Pascal/1000 are packed according to the above rules. Thus a PACKED RECORD, or a PACKED ARRAY which is a field of a PACKED RECORD, may or may not be packed depending on its size. If a variant part is specified it is always word aligned (this is an exception to the rules noted above). Record REC (above) requires one word.

UCSD Pascal packs data into words from low order bits toward high order bits. If an object cannot fit in the remaining bits of the word currently being packed, or if it must be word aligned because it is structured (see below), some remaining high order bits may be unused.

PACKED in UCSD Pascal is only effective for the last occurence of ARRAY in a declaration. Array ARR (above) uses ten words. UCSD Pascal packs by the rules that any scalar object that occupies less than a word will not cross a word boundary, and that any structured object or an object which requires more than a word of storage will be aligned on a word boundary. So in ARR, each element of the PACKED ARRAY [1..8] OF BOOLEAN takes one bit, however each PACKED ARRAY [1..8] OF BOOLEAN is a structured object and so must be word aligned.

The fields of PACKED RECORDS in UCSD Pascal are packed according to the above rules. Thus a PACKED RECORD, or a PACKED ARRAY which is a field of a PACKED RECORD, will not be packed. It will always be word aligned because it is a structured object. If a variant part is specified it is probably word aligned (this is not known for sure). Record REC (above) requires two words in UCSD Pascal.

I/O DIFFERENCES

File Association: Pascal/1000 has two ways to associate an internal Pascal file name with an outside world file. The first way is to place the file in the program parameter list. When the file is opened (RESET, REWRITE, OPEN, or APPEND) the name specified in the appropriate argument position in the run string will be used as the name of the outside world file to be accessed. The second way is to name the file explicitly in the open call. The second parameter to any of the above routines can be a string (constant or PACKED/unPACKED ARRAY OF CHAR) which contains the file name. This form can be used even if the file was specified in the program parameter list and will use the explicit association. If a file is not in the program parameter list and the explicit form of an open routine is not used, then a scratch file will be created (only REWRITE, OPEN, or APPEND may be used initially). If that file is subsequently reopened for reading, the scratch file can be read. Scratch files are normally purged when explicitly closed. See CLOSE below for more information on the actions taken when a file is closed.

UCSD Pascal does not have the program parameter method of associating file names. The predefined files INPUT, OUTPUT, and KEYBOARD are automatically associated with the terminal at program start. RESET can optionally have a second parameter, a string to specify a file explicitly, and REWRITE must always have one.

Interactive Files: There is a basic problem with I/O in Pascal when it comes to dealing with interactive devices. It results from the original definition of the way RESET and READ work on text files. RESET is to perform a GET to load the file window. READ is to use the current window, then use GET as many times as it needs, and then do a GET to leave the window valid when it is finished.

The problems this causes are as follows. By definition the file INPUT (if present) is to be automatically RESET when the program starts. If INPUT is associated with an interactive device this requires that a character be read from it to load the file window. This means the user must type something, even before his program has done anything. The program will be unable to issue a prompt, and it may not even want any input yet. When a program READs the end of line character, READ will want to load the window with another character, so the READ will hang until the next line is typed.

Pascal/1000 and UCSD Pascal take different approaches to solving these problems.

Pascal/1000 does what is known as 'lazy I/O' to solve the problem. This is nothing more than not loading the window until its contents are actually going to be used. So RESET does not perform the initial GET and READ does not do a GET when it is finished doing a READ operation. Instead an internal flag is set that says that a GET has to be done before the window can be used, and any use of the window (explicitly F↑, or with another READ, EOLN or EOF) will cause the GET to occur. To the user program this technique is almost invisible. Programs written assuming the specified behavior of I/O will work as expected. The only known problem involves the EOF function. To determine if EOF has occured from an interactive device on a HP/1000 system, a physical read operation has to be done on the device (a control-d, or carriage return with no data are considered end of file indications). Thus a program may hang on an EOF call until something is typed. If the EOF call is at the top of a loop that prompts the user and then reads data, the prompt will not be seen because the loop can't be entered until after the data is typed. The following restructuring of the loop will solve this problem.

```
REPEAT
    prompt (prompt, 'What now?');
    done := eof (input);
    IF NOT done THEN BEGIN
      read ...
    END
UNTIL done;
```

UCSD Pascal solves the problem by declaring a new kind of text file, the predefined type INTERACTIVE. INTERACTIVE files have the attribute that the GET is not done by RESET and READ does a GET to load the window before accessing it. It has a side effect on the behavior of the EOLN function (see below).

EOLN and EOF: Pascal/1000 EOLN/EOF behave as originally specified with the exceptions noted above. The behavior that just typing a carriage return on a terminal causes end of file is not a function of Pascal/1000 but of the underlying file access routines. It is expected that this inconvenience will eventually be fixed. It can be worked around with something like this:

```
IF eof (input) THEN BEGIN
   reset (input);
   emptyline := true;
END
ELSE BEGIN
   readln (data);
   emptyline := false;
END;
```

UCSD Pascal files of type INTERACTIVE behave differently on EOLN than originally specified. In Pascal/1000, and in the original specification, EOLN becomes true when the last character on the line is read, and the next character will be the EOLN character (which reads as a blank). In UCSD Pascal on an interactive file, EOLN becomes true after the EOLN character (which reads as a blank) has been read, not before. UCSD does not have the EOF problem on INTERACTIVE files because an explicit end of file character must always be typed to cause end of file.

WRITE and WRITELN to text files: Pascal/1000 allows parameters of type BOOLEAN in WRITE and WRITELN to text files. WRITE does not cause data to be written, the data will be buffered until a WRITELN or PROMPT (see Other I/O Intrinsics below) is performed, or the file is closed (and the buffer flushed).

UCSD Pascal does not allow parameters of type BOOLEAN in WRITE or WRITELN to text files. WRITE will cause data to be written, the data is not buffered. The procedure PROMPT is not available (or necessary).

READ and WRITE to non-text files: Pascal/1000 allows the procedures READ and WRITE to be used on files of all types.

UCSD Pascal limits the use of READ and WRITE to files of type INTERACTIVE, TEXT, or CHAR.

I/O DIFFERENCES

Random Access Files: Pascal/1000 provides a number of intrinsic routines to do random access I/O. One of the routines it provides is SEEK. SEEK takes a file and a position. Records are assumed to be numbered from one. SEEK sets the window so that the next GET/READ or PUT/WRITE will access the object at the specified position. Random access I/O cannot be performed on text files. To use a file in random access mode, the file must be opened with the routine OPEN. The routines READDIR and WRITEDIR are provided to perform both the SEEK and the specified I/O. The function POSITION returns the current position in the file. The function MAXPOS returns the largest possible position in the file. An attempt to READ or WRITE beyond MAXPOS will cause a runtime error.

UCSD Pascal provides the routine SEEK to position the file window. SEEK takes a file and a position. Records are assumed to be numbered from zero. SEEK sets the window so that the next GET or PUT (not READ/WRITE, see below) will access the object at the specified position. SEEK can apparently be used on text files. It sets both EOF and EOLN to false. The subsequent GET or PUT will set these conditions as appropriate. An attempt to PUT past the physical end of file will set EOF true.

Untyped Files: Pascal/1000 does not have untyped files.

UCSD Pascal has untyped files which are used when I/O is done totally with the UCSD I/O intrinsics BLOCKREAD, BLOCKWRITE, and IORESULT. They are declared by leaving the 'OF <type>' part off of the file declaration:

```
VAR binfile: FILE;
```

RESET and REWRITE: In Pascal/1000 a RESET closes the file if already open and then opens it for reading at the start of the file. REWRITE closes the file if already open, then clears it out and opens it for writing at the start of the file. APPEND closes the file if already open, then opens it for writing at the end of the file, without clearing out the old contents. OPEN (not allowed with text files) closes the file if already open, and then opens the the file for random access at the start of the file. The second parameter to any of these routines is described above in 'Associating Files'. The third optional parameter is a string which allows specification of certain file attributes. For REWRITE or APPEND on text files the options are 'CCTL' (default) or 'NOCCTL' to specify if the file should have an initial carriage control character prepended on each line (the PAGE and OVERPRINT routines cannot be used with NOCCTL). All files may be opened 'SHARED' (for shared access with up to six other users) or 'EXCLUS' (for exclusive access). If the second parameter is not used, but the third is, the second can be omitted (i.e the two commas appear next to each other).

In UCSD Pascal files may all be RESET, as reset simply marks a file available for I/O. To create an output file REWRITE must be used. If a file is already open a RESET or REWRITE will cause a non-zero IORESULT (see Other I/O Intrinsics) and the state of the file will not change. Files must be closed first and then reopened.

CLOSE: Pascal/1000 has an explicit CLOSE routine that the user may call to cause other than the default close action on a file. Files are normally closed when the block in which they are declared is exited. Files in arrays or records are not closed and must be explicitly closed by the user with CLOSE. The second parameter to CLOSE is an optional string that specifies the disposition of the file. If an actual (as opposed to scratch) file is opened with RESET, REWRITE, APPEND, or OPEN, the normal action of CLOSE is to flush any output buffers (if applicable) and then close the file (so that it cannot be read from or written to). If it is desired to have such a file removed, the option string 'PURGE' can be specified. When a scratch file is closed, the normal action is to purge it. If it is desired to write onto a scratch file and then later read from it, do not explicitly CLOSE the file after writing on it. Simply do a RESET on the file, this will close (but not purge it) and then open it for reading. When the routine in which the scratch file is declared terminates (or CLOSE is explicitly called) the file will then be purged. If a scratch file is to be preserved it must be explicitly closed with the 'SAVE' option string.

UCSD Pascal has an explicit CLOSE routine as well. It is assumed but not known if files are normally closed at block exit. The second parameter to CLOSE is an option word (not a string) that specifies the disposition of the file. Files opened with REWRITE are normally purged (if on disk, not if a printer); to save them the option LOCK should be specified. The option PURGE will remove the file, and the option CRUNCH will LOCK the file at the point of last access (i.e. will truncate)

Other I/O Intrinsics: Pascal/1000 has the following additional I/O intrinsics:

APPEND — opens a file for writing (like REWRITE) but does not erase the old contents. Sets the file window after the last component in the file so that new components can be appended on the end.

LINEPOS — returns the number of characters read from or written to a text file since the last EOLN. It does not include the character in the window.

PROMPT — causes any data output by previous calls to WRITE and any data specified in the PROMPT call to be displayed without a new line being started (the cursor is left at the end of the output line). This call is only allowed for text files, and is of most use when the file is associated with a terminal, although it will work with files. Data written to a text file will be buffered (and not displayed) until a WRITELN, PROMPT, or OVERPRINT is performed, or the file is closed.

OVERPRINT — causes any data output by previous calls to WRITE and any data specified in the OVERPRINT call to be written to the file in such a way that the next line of data written will overwrite it. This call is only allowed for text files, and is of most use when the file is associated with a terminal or line printer, although it will work with files. See PROMPT for information on the normal buffering action for text files.

UCSD Pascal has the following additional I/O intrinsics.

BLOCKREAD, BLOCKWRITE, IORESULT — Used to read and write blocks of data from and to untyped files. Any number of blocks of data can be transferred. A block is 512 bytes. The function IORESULT returns non zero if an error occurred on the last transfer. Both BLOCKREAD and BLOCKWRITE return the number of blocks transferred. If this is not the specified number of blocks, an error has occurred. This type of operation can be performed in Pascal/1000 by declaring variables to hold DCB's and then using FMP routines which are declared EXTERNAL.

UNITREAD, UNITWRITE, UNITBUSY, UNITWAIT, UNITCLEAR - Used to read and write data to low level peripheral devices. UNITBUSY returns status information, UNITWAIT waits for NOT UNITBUSY, and UNITCLEAR cancels an I/O operation. This type of operation can be performed in Pascal/1000 under RTE-IVB using EXEC calls.

DYNAMIC MEMORY DIFFERENCES

Implementation: . Pascal/1000 allows the heap/stack area to reside in the 32K logical address space (the HEAP 1 option, which is the default) or in the EMA area (the HEAP 2 option). Pointers are one word variables in the former case, two words in the latter. The stack grows from lower address to higher addresses, the heap from higher addresses to lower addresses. If the top of heap passes the top of stack a runtime error occurs.

UCSD Pascal keeps its heap in the 32K address space and pointers are one word variables. The stack grows from higher addresses to lower addresses, the heap from lower addresses to higher addresses. If the top of heap passes the top of stack a runtime error occurs.

Dispose: Pascal/1000 implements the DISPOSE routine to deallocate dynamic variables created by NEW so that the space they occupied can be subsequently reused. Both the standard and extended forms of NEW and DISPOSE are implemented. See Chapter 8, Reducing the Size of Loaded Programs, for information on a version of the heap management routines in which DISPOSE does not actually return the space to the free space list.

UCSD Pascal does not implement DISPOSE.

MARK and RELEASE: Pascal/1000 implements MARK and RELEASE to manage the heap in large chunks. If it is known that a group of dynamic variables will be created, used, and then will all no longer be needed, the state of the heap can be MARKed. Then when all the variables are no longer needed, the heap can be RELEASEd back to the state at which it was marked, effectively disposing of all of the dynamic variables at once. However any pointers to these variables are not cleaned up and the user must take care not to dereference them. The argument to MARK may be of any pointer type, and it must be passed unaltered to RELEASE at the appropriate time.

UCSD Pascal implements MARK and RELEASE. The above discussion is also applicable with the exception that the argument to MARK and RELEASE must be a pointer to INTEGER.

Available Memory Information: Pascal/1000 has two routines @GHS1 and @GHS2 which can be called from user programs to acquire information about the state of the heap/stack area. The particular routine depends on the setting of the HEAP compiler option. See Appendix F.

UCSD Pascal has an intrinsic function MEMAVAIL that returns the number of words between the heap and the stack.

INTRINSICS DIFFERENCES

PACK and UNPACK: Pascal/1000 implements the PACK and UNPACK routines. Implicit packing and unpacking occur in assignments and comparisons.

UCSD Pascal does not implement the PACK and UNPACK routines. Implicit packing and unpacked occur in assignments and comparisons.

EXIT: Pascal/1000 does not provided a similar facility. A GOTO can be used to get to a more outer block, but there is no way to return automatically to the statement following the statement that invoked any particular routine earlier in the call chain (see below). Also if any recursive invocations of a routine are active, the environment is not cleaned up on the GOTO.

UCSD Pascal provides the intrinsic procedure EXIT as a way to leave cleanly a series of PROCEDURE or FUNCTION calls. The call EXIT (P) causes control to return to the statement immediately following the the statement which invoked the routine P. This can be several levels up the call chain. For example the main program could call P, P could call Q, Q could call R, R could call S, and S could to an EXIT (P). The effect is that S, R, Q, and P are all exited and control is returned to the main program immediately following the call to P. If EXIT is used to exit a function, the results will be undefined if no value has been assigned to the function before the EXIT call.

Turtle Graphics: Pascal/1000 does not provide anything similar to this feature. Pascal/1000 programs can use graphics routines provided by the system or written in a language with a compatible calling sequence.

UCSD Pascal provides a series of primitive graphics intrinsics. These include CLEARSCREEN, MOVE, MOVETO, PENCOLOR, TURN, TURNTO, and WHEREAMI. They are intended for use on a video display device with graphics capability.

Other Intrinsics: Pascal/1000 does not have any other intrinsics that are not described elsewhere in this document. Pascal/1000 has access to FMP, IMAGE, DS/1000, VIS, RLIB, and other system routines which can be called with either the .ENTR or DIRECT calling sequence if these routines have been declared EXTERNAL in the Pascal source. Pascal/1000 does have user callable library routines which are described in detail in Appendix F of the Reference Manual and briefly noted below.

RSPAR — get run string parameters or the entire run string.
@RDCB — get address of DCB within Pascal file variable.
@RNAM — get the name of a scratch (or any) file.
@GHS1 — get heap/stack information in HEAP 1 program.
@SHS1 — set heap/stack information in HEAP 1 program.
@INH1 — initialize heap/stack information in HEAP 1 program.
@GHS2 — get heap/stack information in HEAP 2 program.
@SHS2 — set heap/stack information in HEAP 2 program.
@INH2 — initialize heap/stack information in HEAP 2 program.
@TIME — get time string, e.g. 'Mon Sep 29, 1980 2:26 pm'.
@SGLD — load a segment.

UCSD has the following additional intrinsics.

GOTOXY — places the cursor at the specified position on the screen.
IDSEARCH — routine used by the Pascal compiler and the PDP-11 assembler.
LOG — returns as a real result the log to the base ten of the argument.
PWROFTEN — returns as a real result the number ten raised to the power of the argument.
SIZEOF — returns the number of bytes allocated to a variable.
TREESEARCH — routine used by the Pascal compiler.

SIMILAR RUNTIME SUPPORT ROUTINES

HALT Procedure: Pascal/1000 has a standard procedure HALT which takes an integer argument. The routine displays

PASCAL HALT: n

where n is the integer passed as the argument. It then makes an effective branch to the end of the main program. Main program files will be closed, but any local files open in any active procedures and functions being terminated by the HALT will not be closed.

UCSD Pascal has a standard procedure HALT which takes no parameters. It causes a non-fatal runtime error and normally the debugger is entered. If the debugger is not loaded a fatal runtime error occurs.

TIME Procedure: Pascal/1000 has a library routine @TIME which takes a single VAR parameter of type PACKED ARRAY [1..26] OF CHAR and returns a time string of the form that appears at the top of the source listing generated by the compiler. Time of day information can be obtained from the RTE-IVB system with an EXEC 11 request. Note that @TIME is not an intrinsic, it must be declared EXTERNAL with an ALIAS to @TIME (which is not a legal Pascal identifier).

UCSD Pascal has an intrinsic procedure TIME which takes two one word integer VAR parameters and returns the high and low order word of the system clock (which is assumed to be a 32 bit quantity which is incremented every 60th of a second). This procedure will not work if the hardware does not support a system clock.

LIMITS DIFFERENCES

Maximum Size of Object Code: A Pascal/1000 program source with a total code and data size in excess of 32,767 words will not compile (a location counter overflow syntax error will occur). In a segmented program, the size of the main plus that of the largest segment must be considered (since they are separate sources the compiler can't detect a potential overflow, but the loader will if it occurs). The particular machine and/or operating system that the program runs on/under will impose its own limits. This can range from 26 to 28 pages (of 1,024 words each). The program as well as Pascal runtime support and system routines must fit within the limit. So there are programs that will compile but which will not load. Other than this there are no restrictions on the amount of code or local data a program, procedure, or function may have.

UCSD Pascal procedures and functions may occupy up to 1,200 bytes of code and have up to 16,383 words of local variables. Total program size limits are not known, but must be less than 32,767 words on standard sixteen bit machines.

Maximum Number of Procedures and Functions: Pascal/1000 imposes no limit except those noted above. Compilation units with large numbers of identifiers will require that the compiler be given a larger workspace, which can affect compile speed, and could possibly be impossible to compile in a given size EMA partition.

UCSD Pascal has a limit of 127 declared procedures and functions per program.

SUMMARY

It has not been my intention in this comparison to rate either UCSD Pascal or Pascal/1000 as "better" or "worse", but rather to simply make clear where the differences lie.

There is currently no "standard" Pascal although there is an ANSI committee working on one (in collaboration with the British Standards Institute and the International Standards Organization). During the development of Pascal/1000 every attempt was made to keep track of the standardization effort and to make Pascal/1000 conform as closely as possible. Most probably Pascal/1000 will not conform exactly to the standard when the standard appears, however we are committed to bring it into conformance with the standard.

At the time UCSD Pascal came into existence there was no such standards effort that they could track. UCSD Pascal has become a sort of de facto standard in the small computer area and they have lobbied to have some of their features in the official standard. One would presume that when the standard appears that SoftTech (the current source of UCSD Pascal) will either make UCSD Pascal conform or will produce a new Pascal product which meets the standard. However the question of the large number of UCSD Pascal programs that will not compile under a "standard" compiler remains unresolved.

Both UCSD Pascal and Pascal/1000 have provided extensions to the language that are specific to their users requirements. In this sense both languages deviate from the standard, and a program which uses any extension of either language may not be easily portable to another Pascal compiler environment. Pascal/1000 provides a compiler option that will cause the use of any extension to be flagged as a syntax error, in an attempt to make the writing (and verifying) of portable programs possible.

ACKNOWLEDGEMENTS

I'd like to thank Linda Siener and Gary Lauber for providing me with some of the documentation on UCSD Pascal, and Ron Mak for making some empirical tests of 'the facts' on his UCSD Pascal system, as well as for his diligent proofreading of this document.

REFERENCES

On HP Pascal/1000

Hewlett Packard Pascal/1000 Reference Manual (92832-90001)
Library Index No. 2RTE.320.92832-90001
Hewlett Packard Company, Data Systems Division, May 1980

Grogono, Peter
Programming in Pascal, With Pascal/1000 (92832-90002)
Addison-Wesley, 1980

On UCSD Pascal

Shillington, Keith Allen and Ackland, Gillian M. (ed.)
UCSD PASCAL System II.0 User's Manual March 1979
Institute for Information Systems,
University of California, San Diego

Bowles, Kenneth L
Microcomputer Problem Solving Using PASCAL
Springer-Verlag, 1977

On Pascal in general

Wirth, Niklaus and Jensen, Kathleen
Pascal User Manual and Report, 2nd edition
Springer-Verlag, 1978

Grogono, Peter
Programming in Pascal (the original)
Addison-Wesley, 1979

Addyman, A. M. and Wilson, I. R.
A Practical Introduction to Pascal
Springer-Verlag, 1979

A NEW FORTRAN INDEPENDENT STUDY COURSE

By Shauna Uher/HP Data Systems Division

The FORTRAN IV independent study course has been updated to include instruction in the new FORTRAN 4X compiler and the EDIT/1000 text editor. The course now consists of eight color videocassettes: the original six videocassettes of the FORTRAN IV course and two additional videocassettes.

This course is fully modular in design, divided by subject into 11 sessions for easy use by first time FORTRAN programmers, and for selective review by previously experienced programmers.

Session	Topic
1	Introduction, Course Organization, History of Programming Languages
2	RTE-IVB procedures, including EDITR
→ 2'	RTE-IVB procedures, including EDIT/1000
3	FORTRAN Character Set, Operators, Expressions
4	Input/Output Procedures
5	Branching, Testing and Looping
6	DO Loops
7	Array Processing
8	Functions and Subprograms
9	Debugging a FORTRAN Program
10	Additional FORTRAN Statements, VIS/EMA Features
→ 11 part 1	FORTRAN 4X Features
part 2	File I/O with FORTRAN 4X

The arrows point out the sessions that were added to the FORTRAN IV course. Session 2' teaches program development with EDIT/1000 HP's new text editor. Either session 2 or session 2' can be viewed depending on the editor used on your system. Once the student is proficient in FORTRAN, session 11 can be viewed to learn the FORTRAN 4X enhancements, such as, the IF/THEN/ELSE block, double integers, and file I/O.

A student workbook is used by each student to supplement the material on the videocassettes. Self-evaluation questions at the beginning of each session allow the student to evaluate the session for content prior to viewing the videocassette, and skip to the next session if he or she feels proficient in the material. Supplemental problems and lab exercises (as well as possible solutions) are provided for hands-on experience.

A person knowledgeable in FORTRAN 4X should be available to assist the inexperienced student when required. An instructor's guide is is provided with each set of videocassettes to facilitate this "advisor" in providing aid to the student. The instructor's guide also provides information on giving a classroom course for group on-site training by the user.

ORDERING INFORMATION

22958C FORTRAN INDEPENDENT STUDY COURSE PRODUCT NUMBER

OPT 001	U-Matic 3/4 inch color videocassettes
OPT 002	VHS 1/2 inch color videocassettes
OPT 003	one student workbook
OPT 004	Betamax 1/2 inch color videocassettes
OPT 010	for update videocassettes only

To order the complete set of videocassettes specify option 001, 002, or 004. List price for this set is $1350.00.

If you already have the FORTRAN IV independent study course, you can order the 2 updated videocassettes by specifying option 010 in conjunction with option 001, 002, or 004. List price for the update package is $350.00.

Both of the above sets include a copy of the updated instructor's guide.

As many student workbooks as required may be ordered as option 003. List price for each workbook is $68.00. A set of overhead slides are separately orderable as part number 22999-90330. List price for this set of slides is $350.00.

Note that for customers who desire off-site training in FORTRAN, this course is being offered at HP Regional Training Centers with systems and experienced instructors available to the student through an intensive five-day schedule.

ATTENTION EUROPEAN USERS!!

Do you feel underprivileged, listless, and lacking in software? The diagnosis is lack of user interaction. The cure is to join a local user group.

User groups now exist in six European countries, and we are all waiting for you to join us.

A local User Group meeting will have on its agenda some or all of the following items, some of which will be of interest to everyone.

User presentation.
H.P. presentation.
Problems session.
"Would'nt it be nice if".
Presentations of user software available at no charge.
Swap shop.
Plus a variety of other items, including news about the International User Group.

In addition to the local user group, some countries have sub- groups covering, for example in the U.K., Image, Microprocessors, and in the near future Graphics.

If you live in Spain, Austria, Luxemborg, Denmark, Sweden, Italy or Finland, then contact your main H.P. office, and they will either put you in contact with another local group, or, why not form a group in your country. The European committee will be pleased to give you advice on how to set up a group and present you with the "How to start a user group kit".

Jean-Louis Rigot
Technocatome TA/DE/SET
Cadarache
BP. 1
13115 Saint Paul les Durance
France
Tel. (42) 25-39.52

Hermann Keil
Vorwerk +Co Elektrowerke
Abt. TQPS
Rauental 38-40
D-5600 Wuppertal 2
W. Germany
Tel. 202-603044

Albert R. Th. van Putten
National Institute of Public Health
Antonie van Leeuwenhoeklaan 9
Pbox 1
3720 BA Bilthoven
The Netherlands
Tel. 30-742344

Graham Lang
Laboratories RCA Ltd
Badenerstrasse 569
8048 Zurich
Switzerland
Tel. (01) 526350

Mike Bennett
Riva Turnkey Computer Systems
Caroline House
125 Bradshawgate
Bolton
Lancashire
U.K.
Tel. 0204-384112

Prof. Tibergen
Vrije Universiteit Brussel
Dienst Informatie Verwerking
Pleinlaan 2
1050 Brussels
Belgium

JOIN AN HP 1000 USER GROUP!

Here are the groups that we know of as of December 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
Arizona	Jim Drehs 7120 E. Cholla Scottsdale, Arizona 85254
Boston	LEXUS P.O. Box 1000 Norwood, Mass. 02062
Chicago	David Olson Computer Systems Consultant 1846 W. Eddy St. Chicago, Illinois 60657 (312) 525-0519
Greenville/S. C.	Henry Lucius III American Hoechst Corp. P.O. Box 1400 Greer, South Carolina 29651 (803) 877-8471
Huntsville/Ala.	John Heamen ED35 George C. Marshall Space Flight Ctr. Nasa Marshall Space Flight Ctr., AL. 35812
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

NORTH AMERICAN HP 1000 USER GROUPS (CONTINUED)

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

OVERSEAS HP 1000 USER GROUPS

London	Rob Porter Hewlett-Packard Ltd. King Street Lane Winnersh, Workingham Berkshire, RG11 5AR England (734) 784 774
Amsterdam	Mr. Van Putten Institute of Public Health Anthony Van Leeuwenhoeklaan 9 Postbus 1 3720 BA Bilthoven The Netherlands
South Africa	Andrew Penny Hewlett-Packard South Africa Pty. private bag Wendywood Sandton, 2144 South Africa
Belgium	Mr. DeFraine K.U.L. Celestijneulann, 300C B-3030 Heverlee Belgium

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. 4/81

Part No. 5951-6111