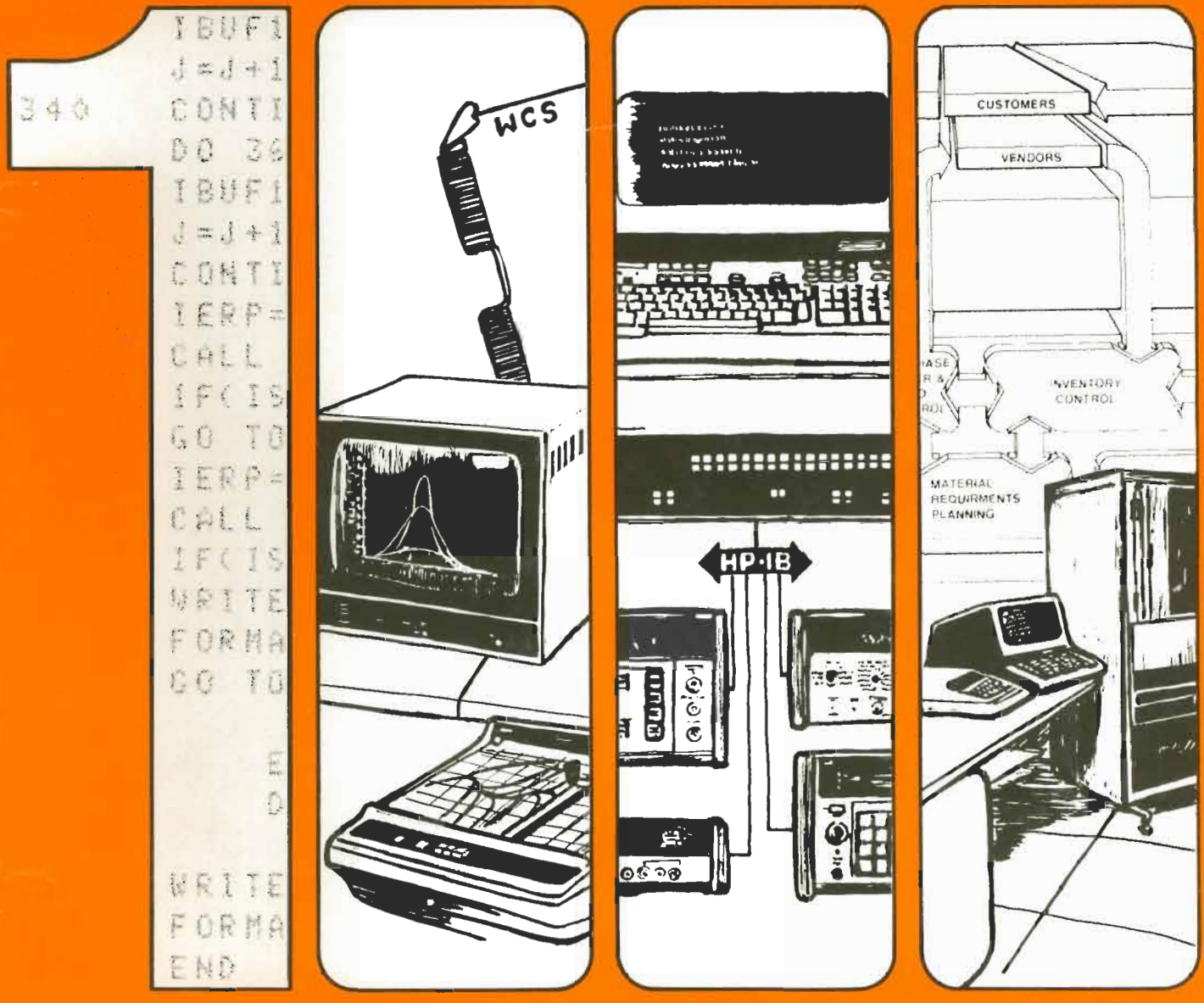
HEWLETT PACKARD

Hewlett-Packard Computer Systems

COMMUNICATOR



1980

Volume IV Issue 3

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HEWLETT-PACKARD
COMPUTER SYSTEMS

Volume IV
Issue 3

COMMUNICATOR/1000

Feature Articles

Departments

EDITOR'S DESK

ABOUT THIS ISSUE

Dear Readers,

I hope that each of you gets as much out of reading this issue as I did assembling it. Even though I am prejudiced, I feel that the articles that you have invested the effort in contributing, have provided the Communicator with a wide variety of excellent articles to publish in issue 3.

For any of our readers that have been intrigued by the intricate graphics that experienced programmers seen to produce so effortlessly, Kathleen Sandifur of Technology Development Corporation has written the first feature article presented in this issue. Kathleen presents four very fundamental concepts of graphics programming in her article in the Computation section. She has included several examples that will, I'm sure, speed the novice on his way toward "first class" graphics output.

The second feature article is provided to us by Shulom Kurtz, a Systems Engineer in HP's office in Englewood Colorado. Shulom's article, "Generating RTE for Pleasure and Performance" provides several hints to make both generations and the resulting systems run faster. Be sure to see the Editor's note at the end of Shulom's article which points out some closely related articles from previous Communicators.

The second article in the Operating Systems category is the continuation of the Links/1000 series by Bob Niland of HP's Lexington Massachusetts office. In this, the fourth in the series, Bob discusses the use of the .ENTR subroutine to pass parameters into assembly language subroutines. Again Bob has been successful in describing some standard procedures (and pointing out some tricks) in language that every level of RTE user can understand. Thanks again to Bob for all of his efforts in producing the Links/1000 series.

The final feature article presented in this issue is entitled, "Multiple Terminal Access to a Real Time Data Acquisition System" by Bradley Ward, an independent consultant in Atlanta Georgia. As the title implies, Bradley discusses concepts relating to I/O in a real time data acquisition environment. I believe that Communicator readers will also find his article to be a good review of how to use class I/O, and SSGA. Hats off to Bradley for completing our versatile and informative quartet of feature articles in issue 3.

As I hope each of our readers is aware, a calculator is awarded for the best feature article, from an eligible contributor in each of the three categories, Customers, HP field personnel, and HP factory personnel. The eligible calculator winners for this issue are listed below:

Best Feature Article From A Customer	**Multi Terminal Access to a Real Time Data Acquisition System** Bradley Ward/Consultant
Best Feature Article From an HP Field Employee	**Generating RTE for Pleasure and Performance** Shulom Kurtz/Englewood, Co.

Thanks to everyone whose effort helped to make this Communicator possible.

The Editor

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

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

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

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

WIN AN HP-32E CALCULATOR!

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

1. HP 1000 Customers;

2. HP field employees;

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

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

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

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

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

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

   OPERATING SYSTEMS
   DATA COMMUNICATIONS
   INSTRUMENTATION
   COMPUTATION
   OPERATIONS MANAGEMENT

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

EDITOR'S DESK

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

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

A SPECIAL DEAL IN THE OEM CORNER

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

IF YOU'RE PRESSED FOR TIME . . .

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

THE MECHANICS OF SUBMITTING AN ARTICLE

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

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

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

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

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

With best regards,

The Editor

A CORRECTION TO A PREVIOUS ISSUE

I received a letter recently from John Pezzano, of HP's El Paso Texas office. John pointed out some misprints in Volume IV Issue 1 of the Communicator. The article that he was referring to was Lyle Weiman's Bit Bucket article on *"How to Prevent DS Monitors from Hogging Partitions"*.

Specifically, John notes that following the call to CNUMD, in line 79 of the FLUSH program listing, DEF * +13 should actually be a DEF * +3. Three lines further in the code the source and destination addresses are loaded into the A and B registers. The labels that are given however (MSF1 and MSF2) are incorrect. The instructions should appear as a LDA @MSF1 and LDB @MSF2. Finally, the astute reader will notice that six lines from the end of FLUSH, the instruction that reserves a memory location to be used by NPART has been shifted to the right. If typed in exactly as it appears, an error will occur. The label NPART must be moved to begin in column 1 and the NOP instruction should appear in column 7 like the other instructions in the code.

Thanks John for pointing out these errors. The Editor apologizes for any inconvenience that this error may have caused our readers.

The Editor

EDITOR'S DESK

LETTERS TO THE EDITOR

Dear Editor,

My new HP 1000 was just installed along with the 2001 revision of software. In volume three, issue five of the Communicator the problem of SRQ.T appearing as an undefined external was discussed. One "fix" described, was to install the 1940 revision of the library %IB4A. My system contains the 2001 revision of this library, yet I still get SRQ.T as an undefined external. Any explanation?

Sincerely,

Dr. James F. Elder
Research Department
R. J. Reynolds Tobacco Company
Winston-Salem, N.C.

Dear Dr. Elder,

I will assume that the undefined externals that you speak of are occurring when you load a program which you have written, and not in your RTE-IVB generation. (SRQ.T will still show up as an undefined external if %IB4A is loaded in a generation, but the BASIC memory resident library isn't).

Since I'm assuming that you are loading a program, I will also assume that you are writing the program in FORTRAN, and not BASIC. This is an important issue, because the subroutine calls that are used to schedule programs as a result of SRQ are different for the two languages. In a FORTRAN program, a call should be made to "SRQ" to identify the program that should be scheduled when SRQ is asserted. In BASIC the subroutine "SRQSN" is used to set a BASIC trap.

If SRQSN is called from FORTRAN, and the BASIC memory resident library is not in the system, SRQ.T can be expected as an undefined. Furthermore, if it got satisfied, totally unexpected results would occur.

Assuming once more that the correct subroutine call, SRQ, is being used, there is only one other way that SRQ.T could come up as an undefined external. When your HP-IB program is loading, if the entire library, %IB4A is relocated, instead of just the modules out of it that are needed, the SRQSN subroutine will be appended to your program, eventhough it will never be used. As described above, this subroutine is the culprit; it makes an external reference to SRQ.T.

How could the entire library be relocated instead of just the needed modules? The only way this could occur would be for the user to command the loader to "RE,%IB4A" (RElocate the module %IB4A). If %IB4A is generated into the system, the user needs only to have the loader "SEA" (search the system library, and attempt to satisfy all backward references). An SEA will cause only the routines that are necessary to be relocated. The EN command will also cause the system library to be searched to satisfy any undefined externals. If the %IB4A library is not generated into the system, the user will need to "SEA,%IB4A" before typing "EN". Again, if "RE" is used, the entire library will be appended to the program, wasting program space, and pulling in extra subroutines (like SRQSN) which can cause problems (like undefined externals).

Hopefully the information above will help you determine why SRQ.T is showing up as an undefined external, and how to eliminate this problem. If this explanation is not satisfactory, please contact me.

Sincerely,

The Editor

Dear Editor,

Many thanks to Todd Field for his article, "Easy Forms For The 2645A", in Volume III, Issue 5 of the Communicator 1000. I intend to use these ideas extensively for data entry. However, I found some errors in one of the listings which could be disastrous to the unsuspecting user.

In the program F2645, change the following;

Lines 70 and 71 should be changed

```
          FROM                                    TO

CALL REIO(3,LU,22B,0)                  CALL REIO(3,LU+2200B,0)
CALL REIO(3,LU,25B,0)                  CALL REIO(3,LU+2500B,0)
```

Also, line 143 in the listing should be replaced by the following two lines of code;

```
CALL REIO(3,LU+2200B,nnnn)
CALL REIO(3,LU+2500B,0)
```

The nnnn parameter above represents the time-out value, in tens of milliseconds, to which the terminal is to be reset.

If the schemes that Todd describes are to be used with very large or complex form lines, the dimension for INBUF should be increased from 80 to about 400. If this change is made, the following changes will also have to be made.

Line 93 should be changed

```
          FROM                                    TO

DO 605,I=6,80                          DO 605,I=6,400
```

Line 99 needs also to be changed

```
          FROM                                    TO

CALL REIO(1,LU+1000B,INBUF(6),75)  CALL REIO(1,LU+1000B,INBUF(6),395)
```

Another handy modification that I made was a change to the WRITE and FORMAT statement, starting on line 51, to that the softkey description would be output to the printer. Line 62 can then be omitted. For my application, I also expanded the softkey definitions. I replaced lines 51 through 62 with the following:

```
            WRITE(LU,410)
      C^Y^K
^&13V  410   FORMAT("^c^h^J^&f1a1k 6L^&dB^[",^K
^&13V       >              "^&f1a2k 6L^]^&d@",^K
^&13V       >              "^&f1a3k 8L^&dB^[^7",^K
^&13v       >              "^&f1a4k 8L^]^8^&d@",^K
^&13v       >              "^&f1a5k 8L^&dB ^[^6",^K
^&13v       >              "^&f1a6k 8L^]^8^&d@")^K
^&13v  C^Z
      C
            WRITE(6,411)
      411   FORMAT(//" f1 - START UNPROTECTED, ALPHA/NUMERIC FIELD",
           >       //" f2 - STOP UNPROTECTED, ALPHA/NUMERIC FIELD",
           >       //" f3 - START UNPROTECTED, NUMERIC FIELD",
           >       //" f4 - STOP UNPROTECTED, NUMERIC FIELD",
           >       //" f5 - START UNPROTECTED, ALPHABETIC FIELD",
           >       //" f6 - STOP UNPROTECTED, ALPHABETIC FIELD",
           >      ///" FIVE CONSECUTIVE BLANK LINES WILL BE",
           >         " INTERPRETED AS THE END OF THE FORM"/,"1")
```

In the subroutine FFORM, the following changes must be made:

Omit line 39.

Make the following changes to lines 45 through 47.

```
                FROM                                    TO

200   CALL REIO(2,LU,HMULK,-5)          200   CALL REIO(3,LU+2200B,0)
      CALL REIO(3,LU,25B,0)                   CALL REIO(3,LU+2500B,0)
      CALL REIO(1,LU,IDUM,1)                  CALL REIO(2,LU,HMULK,-5)
                                              CALL REIO(1,LU,IDUM,1)
```

Omit line 55.

Replace line 67 with the following code:

```
                FROM                                    TO

      CALL REIO(3,LU,25B,0)                   CALL REIO(3,LU+2200B,nnnn)
                                              CALL REIO(3,LU+2500B,0)
```

As above, nnnn should be set to the time-out value for the terminal, in tens of milliseconds.

If INBUF has been increased to 400 in F2645, then this variable should also be dimensioned to 400 in FFORM. In this case, the following code on line 33 must also be changed.

```
                FROM                          TO

  CALL READF(IDCB,IERR,FBUF,80,ILN)     CALL READF(IDCB,IERR,FBUF,400,ILN)
```

After the changes described above are made, the enhanced programs should perform properly. Remember that the information returned in INBUF by the subroutine FFORM is in ASCII. If other types of data are desired, such as real numbers, the buffer must be converted, using a call to CODE. This program is described briefly in the DOS/RTE Relocatable Library manual. To understand the use of CODE, consider the following example.

Suppose that I have set up a form where there are two identical lines, each having two fields. The first field is alpha/numeric, ten words long, and the second field is numeric (real), three words long. The total number of words for both lines is 26. After calling the subroutine FFORM, to display and read the screen, it will be necessary to convert the two numeric fields from ASCII to Real data. To do this, the following sequence should be used:

```
      CALL CODE
      READ(INBUF,100) (INBUF(J),J=1,10),ANUM(1),
     >                (INBUF(J),J=14,23),ANUM(2)
100   FORMAT(2(10A2,F6.2))
C
C  CALCULATIONS - ANUM ARRAY
C
      ANUM(1)=ANUM(1)*2.0
      ANUM(2)=ANUM(1)+ANUM(2)
```

The call to CODE and the READ are considered to be one FORTRAN statement, written in two lines. No statements may be between them. INBUF is unchanged, since it was in ASCII before the call. However, ANUM was converted from ASCII to a Real array, according to the F6.2 specification of the FORMAT statement. After any desired calculations using ANUM are complete, it may be written back to the screen, but not before being converted into ASCII by code similar to the following.

```
      CALL CODE
      WRITE(INBUF,110) (INBUF(J), J=1,10),ANUM(1),
     >                 (INBUF(J), J=14,23),ANUM(2)
110   FORMAT(2(10A2,F6.2))
C
C
C
      CALL REIO(2,LU,INBUF,26)
```

Since the terminal is still in FORMAT MODE, it is important that the exact number of words be written to the screen, in this case, 26.

One final note. It is best not to try to use your system console for any of these programs, because you will get log messages written into your form. A simple routine involving a call to LUTRU will prevent accidental use of the system console.

Sincerely,

W. J. Runkle
Union Carbide Corporation
Indianapolis In.

Dear Mr. Runkle,

I am happy to hear that you have found Todd's article to be so useful. Thanks for pointing out the problems you found in the code to other readers, and also letting everyone know how to make some enhancements that that seem to make the program more flexible.

Sincerely,

The Editor

HP-IB COMMUNICATIONS OVER DS-1000

Daniel G. Antzoulatos/New Mexico State University

Those readers familiar with the HP-IB will agree how versatile an interfacing concept it is; not only from a hardware standpoint, but also from a software one. HP software supports HP-IB communication in various ways allowing for a great deal of flexibility. This may, unfortunately, present a problem in deciding on a particular programming procedure or "style" that can generally be used for all needs. This report describes, briefly, the different aspects of HP-IB communication, while focusing on two special procedures :

- HP-IB compatible DS-1000 calls, and
- operator control of HP-IB.

Let's examine a particular system so that we may have some relative guidlines which to follow. Consider a system which consists of a network of RTE-IV and RTE-M computers, interconnected via DS-1000 links. Almost every node (computer system) has certain HP-IB devices which need to be controlled not only locally, but also from a remote node. For the most part, the HP interface bus dialogue will take place only between the computer and one listener. Nevertheless, we should not eliminate the capability, neither through software nor through hardware, of supporting requests for inter-device communication, and defining multiple listeners.

Thus, our goal is to define a procedure which will enable us to program an HP-IB device remotely, via a DS-1000 link.

ADDRESSING MODE

One of the first decisions anyone using HP-IB has to face, is whether to use "direct" or "automatic" addressing. We can use either one exclusively or allow for the mutual existence of both types of addressing. Let's take a look at the advantages and disadvantages of each mode.

Direct Addressing

Advantages:

— The user has total control of the interface bus, allowing for computer to device(s) and inter-device communication.

— Only one logical unit number needs to be used per bus.

— The devices are accessed through the bus LU and their own switch selectable address.

— Actual I/O transactions are faster and more efficient than comparable transactions using automatic addressing. (Although, in some cases, data buffer massaging may considerably slow down the overall communication).

Disadvantages:

— Worrying about control of the bus complicates the programmer's job.

— Remote direct addressing is not a straightforward task.

Automatic Addressing

Advantages:

— The driver takes care of all bus protocall.

— Since each device is assigned an LU, HP-IB communication can be accomplished with standard READ and WRITE contexts and formatting methods, or any other LU based operation, i.e. operator commands.

Disadvantages: — A logical unit number must be assigned to each device on the bus. (When automatic addressing is to be used exclusively, you need not assign an LU to the bus itself)

— Communication can only take place between the computer and one addressed device.

Mixed Addressing Configuration

Advantages: — Allows both types of addressing.

Disadvatages: — In addition to assigning an LU to each device, one must also be assigned to the bus itself.

As one can see, availability of logical unit numbers is a primary factor in determining an addressing method. The number of LU's to be assigned for use by HP-IB instruments may be decided during system generation time, and if necessary, modificatons can be later made on line.

Assuming that we have enough LU's for mixed addressing, let's concentrate on automatic addressing, with the assurance that we can perform direct addressing if the need arises.

Automatic addressing allows us to operate either from within a user program, or from within the file manager via operator commands. First let's take a look at the more 'proper' environment of the two.

REMOTE COMMUNICATION

All of the programs that we are going to consider will be written in FORTRAN. Programs written in BASIC also support HP-IB communication, but they are limited to local node transactions. That's exactly the limitation we encounter when using WRITE/READ, EXEC calls or the HP-IB library routines such as RMOTE,CLEAR,CMDW,etc.. The standard HP-IB software was not written with remote node communication in mind.

The best alternative is DS-1000 software which in turn, as we shall see later, was not written with HP-IB dialogue in mind. Therefore we must make do with DEXEC calls and READ/WRITE statements which are preceded by a DNODE call. This latter call allows the redefinition of the destination node.

Plain data transactions can easily be accommodated by either the combination of the DNODE and READ/WRITE statements, or by DEXEC I/O calls. For example:

```
          CALL DNODE(NODE)
          READ(LU,*) IBUF
```

or

```
          CALL DEXEC(NODE,ICODE+100000B,LU,IDBFR,IDLNG [,ICBFR,ICLNG])
          GO TO error handling routine
    label CONTINUE
```

Where:

NODE	node where the HP-IB device is located
ICODE	Function code (1 - Read,2 - Write)
ICODE+100000B	When errors are encountered, the statement following the DEXEC call will be executed. Otherwise it is skipped.
LU	logical unit number of the HP-IB device logically or'ed with an indication of the type of data transfer that is to be made
IDBFR	data buffer address
IDLNG	data buffer length
ICBFR	command buffer (allowed only for local calls; when used, 10000B must be added to LU)
ICLNG	command buffer length

Although the READ/WRITE statements are cleaner, they do not provide us with an error detection capability as do the more cryptic DEXEC calls. This capability is especially essential for remote calls, where system console messages are not accessible. Thus we choose DEXEC calls for our data transactions.

Notice that the ICBFR can be used in a direct addressing mode to send commands to a local bus. However, when talking to a remote node, the parameters [ICBFR,ICLNG] cannot be used. Thus, DEXEC is really not a true analog of EXEC, but was born handicapped - not able to perform remote HP-IB 'direct' addressing. However, this is of little consequence to us, since our emphasis is on 'automatic' addressing.

Man cannot survive in the real HP-IB world on automatic data transactions alone. Most devices, but not all, require the HP-IB bus REN (remote enable) control line to be activated before any data can be accepted by the device. This can easily be accomplished by an HP-IB library RMOTE call for local devices, but the same does not apply to remote devices. In this case, we can use the I/O control capability of DEXEC (ICODE=3) to do the job. This can be done as follows:

```
CALL DEXEC(NODE,3+100000B,LU+1600B)
```

where:

NODE and ICODE are the same as those described above.
LU is either the HP-IB bus lu or the device lu.

The 1600B added to the LU is a function code signaling the driver to issue a command to the bus to activate its REN line. There are other such function codes defined in the manual for DVR37, the driver for the HP 59310B interface bus. For example, given the generic call:

```
CALL DEXEC(NODE,3+100000B,LU+XX00B)
```

where:

LU is the HP-IB device LU

then the following values for XX will perform the indicated function.

00 — Selected device clear
01 — Issue EOR
06 — Dynamic device status
11 — Device line spacing
16 — Set REN true
20 — SRQ schedules program in IPRAM
21 — Disable function code 20 IPRAM
25 — Configure driver word, this device only
27 — Clear driver configuration, this device only

If however LU is the interface bus LU

then XX will cause the functions below to be performed:

00 — General bus clear
01 — Issue EOR
06 — Dynamic bus status
16 — Set REN true
17 — Set REN false
25 — Configure driver word
27 — Clear driver configuration
30 — Parallel poll

As you can see, the function codes depend a great deal on the LU. One may wonder if the function of one LU definition can be used to perform an analogous job in another LU definition. For example, can we use XX=17 when LU is the device LU? The answer is, unfortunately, no! This is quite inconvenient if you only readily know the device lu and not the bus lu. You cannot disable the REN line, for instance, without knowing the bus lu.

OPERATOR CONTROL

If you're from the old school of non-standard device interfacing, you may still carry the idea that the only way to talk to a specialized device is through a program that issues a call to a special driver, which controls an interface card like a 16 or 32 bit duplex, through which the data transactions take place. In this case the data format is unique according to the device. But with HP-IB remember, the information is transferred one byte at a time, either in ASCII or binary form. Thus, you are able to use READ/WRITE statements to perform these transactions, the same as if you were talking to a line printer or CRT.

In addition, FMGR provides us with operator commands with which we can transfer information to and from non-disc devices that understand our one byte formats. If we can use these operator commands with say, a line printer or CRT, why not with 'auto-addressable' HP-IB devices? Well, we can! Of course this is a "quick and dirty" way to communicate, but one worth investigating.

For instance, to send data from your terminal to a device (devlu), just use the FMGR 'DU' command

```
:DU,,devlu
data.....
data.....
'control D' or 'RETURN'
:
```

You just transferred a command string to the device in the same way it would have been transferred if you had used a WRITE statement in a FORTRAN program.

In gathering data from a device, it may be convenient to use the LI command. For example, if you want 100 samples from an HP-5340 counter whose lu is 26, and this data is to be stored in an existing file DATA1::10, then simply type:

```
:LL,DATA1::10
:LI,26,,1,100
```

Not only do you get exactly 100 samples, but they are also numbered consecutively, which facilitates plotting. If you have an HP-2648A graphics terminal, you can directly plot in real-time, without having to write one piece of software!

But, you say, it was previously mentioned that we need more than just data transcations. How do we perform control functions as we did with DEXEC ? Fortunately, there is an operator command "CN" which provides us with this capability, i.e.

```
:CN,LU,XXB
```

Where LU and XX are the same as those defined above for the DEXEC calls. For example:

```
:CN,26,16B
```

will activate the REN line for LU 26.

Since remote operator commands are supported by REMAT, we can also talk to remote devices. Caution though, should be exercised as some of the operator commands are not fully supported by REMAT. For example, with the "LI" in REMAT, you cannot specify the number of line to list, thus you are not able to properly terminate a data acquisition sequence.

By no means should you do any serious type of device controlling via operator commands. Nevertheless, when you need a quick check-out of a device, this operator approach comes in quite handy.

Until either DS-1000 calls better facilitate remote HP-IB communication, or conversely that the HP-IB library includes calls that can access remote nodes, the suggestions above should prove quite helpful.

PUT YOUR 264X TERMINAL DISPLAY ON PAPER

David Liu/Hughes Aircraft Company

How many times have you had to copy information from the display of your CRT to paper by hand? Did you ever wish that you could get a hard copy of exactly what is on the screen? Listed below is a simple program to do just that. The program reads the display memory from a 264x terminal and dumps this information to the line printer.

HOW DOES IT WORK?

1. Pick up the Logical Unit numbers of the terminal and line printer as they have been passed by the user in the run string.
2. Enable block mode on the terminal.
3. Tell the terminal driver (DVR05) that the terminal configuration has been updated.
4. Read the display memory from the terminal.
5. Write the diaplay memory to the line printer.
6. Disable block mode on the terminal.
7. Tell the driver that the terminal configuration has been updated again.

EXAMPLE

The run string for PSCRN is shown below. Note that either of the two input parameters may be defaulted. The default terminal LU is 1. If the program is operating in an RTE-IVB environment with session monitor this will always be the terminal from which the program was run. If the program is not running in a session environment, the display on the system console will be printed. The default LU for the print device is LU 6.

```
RU,PSCRN,1,6
```

CONSTRAINTS

There are two constraints that PSCRN works under. The software constraint is that PSCRN can only dump screens from DVR05 terminals (not DVR00). The hardware constraint is really more of a prerequisite. The printed circuit board 02640-60123 inside the terminal must have switch D open to enable page strapping.

A listing of the program is provided below:

```
FTN4,L
      PROGRAM PSCRN
C*****************************************************************
C                                                                *
C      SOURCE PROGRAM: &PSCRN                                    *
C      OBJECT PROGRAM: PSCRN                                     *
C                                                                *
C      PROGRAMMER:     DAVID LIU                                 *
C      LAST REVISION:  03-18-80                                  *
C                                                                *
C      PURPOSE:        reads the display memory and              *
C                      transfers it to the line printer          *
C                                                                *
C*****************************************************************
           INTEGER  BLOCK D    (3)
           INTEGER  BLOCK E    (3)
           INTEGER  BUFFER     (8192)
           INTEGER  CRT
           INTEGER  ENTER
           INTEGER  HOME
           INTEGER  I
           INTEGER  J
           INTEGER  K
           INTEGER  LP
           INTEGER  OUT BUF    (70)
           INTEGER  PARM1
           INTEGER  PARM2
           INTEGER  PARM3
           INTEGER  REG        (2)
           INTEGER  REGA
           INTEGER  REGB
           INTEGER  TERM        (5)
           INTEGER  TST WD
           INTEGER  WD LNG
     C
           EQUIVALENCE (REG(1),REG A)
           EQUIVALENCE (REG(2),REG B)
           EQUIVALENCE (TERM(1),CRT)
           EQUIVALENCE (TERM(2),LP)
     C
           DATA     BLOCK D   /2H%&,2Hk0,2HB /
           DATA     BLOCK E   /2H%&,2Hk1,2HB /
           DATA     BUFFER    /8192*2H  /
           DATA     ENTER     /2H%d  /
           DATA     HOME      /2H%h  /
           DATA     I         /0/
           DATA     J         /0/
           DATA     K         /0/
           DATA     LP        /0/
           DATA     OUT BUF   /70*2H  /
           DATA     PARM1     /0/
           DATA     PARM2     /0/
           DATA     PARM3     /0/
           DATA     REG       /2*0/
           DATA     REG A     /0/
           DATA     REG B     /0/
           DATA     TERM      /5*0/
           DATA     TST WD    /0/
           DATA     WD LNG    /1/
```

```
C
C*********************************************************************
C                                                                    *
C                    M A I N     P R O G R A M                       *
C                                                                    *
C*********************************************************************
C     get terminal's logical unit number
      CALL RMPAR(TERM)
      IF (CRT .EQ. 0) CRT=1
      IF (LP  .EQ. 0) LP =6
C     enable block mode on the crt
      REG = EXEC (2,1400B+CRT,BLOCK E,-5,PARM1,PARM2)
C     update crt configuration by I/O control
      CALL EXEC  (3,2500B+CRT,PARM1)
C     home the cursor on the crt
      REG = EXEC (2,2000B+CRT,HOME,1)
C     send an enter command to the crt
      REG = EXEC (2,2000B+CRT,ENTER,1)
C     read display memory to program buffer
      CALL EXEC (1,3000B+CRT,BUFFER,8192)
C     find the length of the buffer read
      DO 3000 I=16384,1,-1
         TST WD=0
         CALL SMOVE (BUFFER,I,I,TST WD,2)
         IF (TST WD .EQ. 15B) 2000,3000
 2000       WD LNG=(I+1)/2
            I=0
 3000    CONTINUE
C     write program buffer to line printer (assume to be LU  6)
      REG = EXEC (2,2000B+LP,BUFFER,WD LNG)
C     disable block mode on the crt
      REG = EXEC (2,1400B+CRT,BLOCK D,-5,PARM1,PARM2)
C     update crt configuration by I/O control
      CALL EXEC  (3,2500B+CRT,PARM1)
      END
      END$
```

GRAPHICS FUNDAMENTALS

Kathleen Sandifur/Technology Development Corporation

INTRODUCTION

To effectively use any graphics system, a person must first gain a conceptualization of the functions universal to all graphics systems. Often for the graphics neophyte, the only readily available method for mastering these concepts is to attack a dissertation of incomprehensible detail. This can be as frustrating as reading the fine print of your insurance policy, in that it is "hard to see the forest for the trees".

In this article, the author puts a few trees in their proper perspective. Four subroutines of the HP GRAPHICS/1000 software package are singled out: WINDW,LIMIT,VIEWP,and SETAR. While this terminology may be unique to a particular software package, its concepts are universal to all graphics packages. As a vehicle for conveying these concepts, the application program LOGO is documented.

This program incorporates the four subroutines for the purpose of allowing easy manipulation of size, shape, and positioning of a logo. By following the implementation of the four subroutines and the explanation of results related to parameter changes, it is presumed that the uninitiated reader can gain a quick and easy grasp of the graphics function.

The underlying objective of all graphics systems is to capture an image, possibly manipulate it in some manner, and then project it to another location or surface. The image must first be presented to the graphics system. For the purpose of the sample program, this was accomplished by sketching a logo on an arbitrary sheet of graph paper; approximating this sketch with straight line segments; and tabulating the coordinates for the end points of these segments. (See figure 1). (The coordinates were calculated by arbitrarily setting the X and Y axes to correspond to the horizontal and vertical lines of the graph paper.) This process is a form of "digitization", and is only one of the many methods available. All have the objective of conveying the information in a form recognizable to the graphics system. The entry of the X and Y coordinates representing the logo occurs between lines 33 and 101 of the sample program.

Once the image is made available to the system, the process of capturing the image, manipulating it, and projecting it, involves four areas of conceptualization:

1. Determine the boundaries surrounding the image that is to be captured (WINDW).

2. Set the limits of the device to the boundaries of the paper or transparency to be utilized (LIMIT).

3. Determine the boundaries as to where the image is to be projected on the paper or transparency (VIEWP).

4. If it is necessary or desirable to ensure that the projected image is not distorted, then the window surrounding the image, and the viewport on the projecting surface must both have the same width/height ratio (SETAR).

WINDW

When the graphics system receives the digitized representation of the image, it requires a frame of reference to designate where the image to be captured is located. To generate this reference frame or *window*, the WINDW subroutine is invoked. The general form of WINDW is:

```
CALL WINDW(IGCB,X1,X2,Y1,Y2)
```

where (X1,Y1) designates the lower-left corner of the rectangular window and (X2,Y2) designates the upper-right corner. Since this rectangle is to frame the image, or a portion of the image represented in the digitization process, then it follows that the parameters for the WINDW subroutine must be generated from the same X – Y axes, same units, and same origin as was used in the digitization process (those arbitrarily established by the graph paper for the case in point). The setting of the WINDW parameters in line 29 of the sample program to:

```
CALL WINDW(IGCB,0.,355.,0.,130.)
```

specifies that the lower-left corner of the rectangular window is zero units in the X direction and zero units in the Y direction (at the origin on the graph paper). Also, the upper-right hand corner of the window is at 355 units in the X direction and 130 units in the Y direction (far right and middle of the graph paper). Since the window encompasses the entire logo (TDC), the captured image for graphics manipulation will be the entire logo. If however, the window had been specified by:

```
CALL WINDW(IGCB,200.,355.,0.,130.)
```

then the window would only frame the right portion of the logo ("C") and only this image would be available for graphics manipulation (see figure 1).

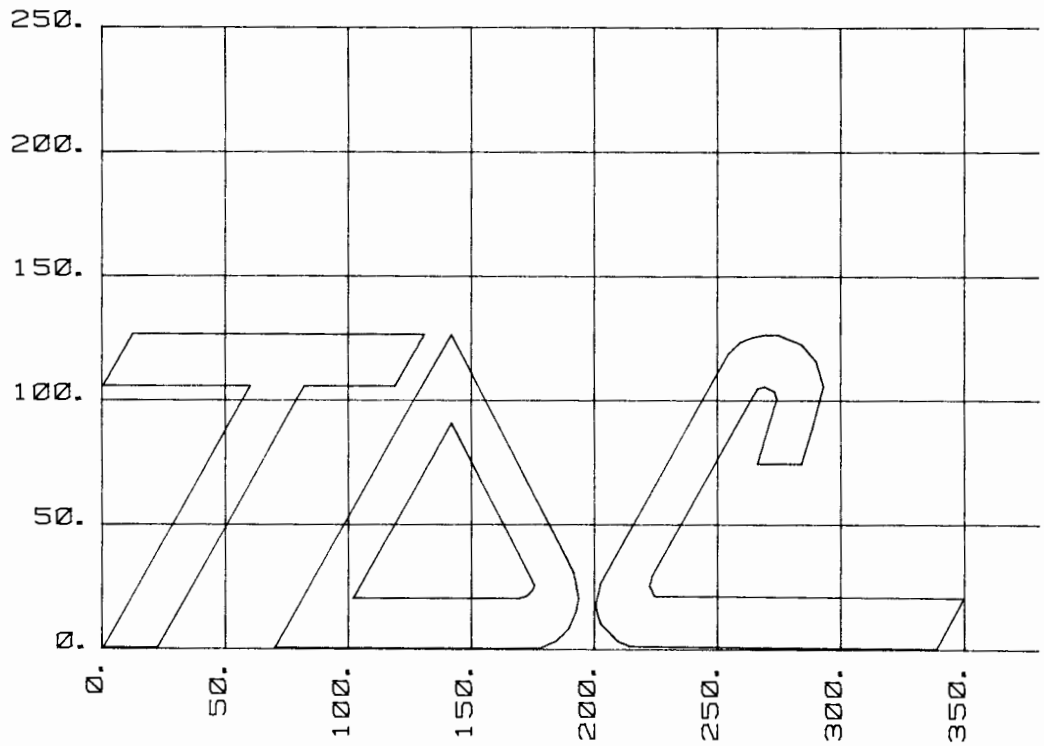


Figure 1.

LIMIT

The LIMIT subroutine defines the physical view surface on the device, or in other words, the surface on the device within which all graphics must occur. After deciding on the size of the paper or transparency desired, the respective width and height dimensions are used to delimit the physical view surface via the LIMIT subroutine. The general form of LIMIT is:

```
CALL LIMIT(IGCB,X1,X2,Y1,Y2)
```

with the X and Y units specified in millimeters. As a result, all graphics must now occur within an area bounded in the horizontal direction from X1 mm to X2 mm and in the vertical direction from Y1 mm to Y2 mm with the origin corresponding to the lower-left corner of a device's plotting surface.

In the sample program, a logo is to be projected to a 350 mm by 280 mm sheet of paper. Line 26 of the program would translate these dimensions to LIMIT parameters as follows:

```
CALL LIMIT(IGCB,0.,350.,0.,280.)
```

To further illustrate the use of the LIMIT subroutine, if it is desired to confine graphics to an 8.5" by 11" (215 mm by 280 mm) area, then set LIMIT parameters as follows:

```
CALL LIMIT(IGCB,0.,215.,0.,280.)
```

VIEWP

Within the view surface set by LIMIT, the option exists for restriction of the projection of the image selected by WINDW to a desired area. Should the logo be spread across the entire paper or transparency, or should it be confined to only a small area? A *VIEWPORT* designates a rectangular portion of the view surface to which the image within the window is to be mapped. The VIEWP subroutine defines the positioning of the viewport. Its general form is:

```
CALL VIEWP(IGCB,X1,X2,Y1,Y2)
```

where (X1,Y1) designates the lower-left corner of the viewport and (X2,Y2) designate the upper-right corner. The X and Y axes correspond to the lower edge and the left edge respectively, of the view surface designated by the limit call.

The units for X and Y vary according to the ratio of the width to the height, or "aspect ratio", of the view surface. If the aspect ratio, AR, is greater than 1, then the horizontal length of the view surface corresponds to 100*AR units and the vertical length corresponds to 100 units. If AR is less than 1, then the vertical length corresponds to 100/AR units and the horizontal length to 100 units. In the instance when CALL LIMIT is not initiated, the view surface defaults to the limit of the device, which for the HP 9872A, for example, has an aspect ratio of 1.52. Therefore, to position the viewport to cover the upper-right quadrant of the view surface, set VIEWP as follows:

```
CALL VIEWP(IGCB,76.,152.,50.,100.)
```

The entry of the viewport parameters in the sample program occurs at line 28.

SETAR

At this point in the graphics explanation the process can be visualized as taking a snapshot and projecting the captured image onto a screen. Everything within the rectangular window is mapped through the rectangular viewport for positioning on the viewing surface, which itself has been delimited via the LIMIT subroutine.

Ifhe rectangular window and the rectangular viewport both have the same shape (that is the ratio of their width to height or *aspect ratio* is the same), then the image can be transferred point for point without any distortion. In other words, geometric figures will not be distorted. A circle will remain a circle and angles between intersecting lines will not change. Only the scale or relative size of the image will change. If the aspect ratio of the window is not the same as that of the viewport, then obviously a point for point mapping of the image in the window, to the image projected on the view surface will be distorted. A circle will become an ellipse and the angle between intersecting lines will be changed. To alleviate this problem, when coverage of the largest possible portion of the selected view surface is desired, the subroutine SETAR (set viewport aspect ratio) is utilized. The general form for SETAR is:

```
CALL SETAR(IGCB,AR)
```

To implement SETAR, the aspect ratio for the window is calculated by dividing its width by its height. The aspect ratio, AR, for the window encompassing the entire logo is 2.73 (355/130 = 2.73). This value is incorporated at line 27 of the sample program as follows:

```
CALL SETAR(IGCB,2.73)
```

The result is that, instead of a point for point mapping of the image in the window to the viewport, the mapping is now from the window to a reconfigured viewport. This reconfigured viewport has the aspect ratio coded into SETAR. This new viewport is shrunk in size such that it just fits inside the old viewport, while maintaining an aspect ratio corresponding to the window. After shrinking the new viewport to fit inside the old viewport, there will be unused area along one dimension of the old viewport. The new viewport is centered along this dimension. Therefore, the image in the window will be mapped undistorted, to an area within the originally specified viewport, centered along one dimension and totally filling the other dimension. This provides the largest undistorted projection of the image onto the delimited view surface without undue calculation being required for viewport positioning.

PARAMETER CHANGES

Thus far, a brief and simplified explanation of each of the four subroutines has been presented and the location of their implementation in the LOGO program specified. The following examples will document how individual changes to the subroutine parameters correlate to output changes from the LOGO program.

1. Set a window to encompass the logo sketch; and default LIMIT, SETAR, and VIEWP parameters (comment out lines 26-28).

```
CALL WINDW(IGCB,0.,355.,0.,130.)
```

The resultant output is a recognizable, although distorted, projection of the logo covering the total viewing surface (see figure 2). The projection covers the total viewing surface because of the default mode for LIMIT and VIEWP. Since a CALL to LIMIT was not initiated, the viewing surface defaults to the physical limits of the device. For the HP9872A, this is equivalent to the following at line 26:

```
CALL LIMIT(IGCB,0.,380.,0.,250.)
```

The viewport must cover the physical limits of the device, for the projection to cover the entire viewing surface. This also occurs by default since a CALL VIEWP was not executed. For the HP9872A this is equivalent to entering the following at line 28:

```
CALL VIEWP(IGCB,0.,152.,0.,100.)
```

The entire logo is projected because the window was chosen of the appropriate size, in units corresponding to those units which the coordinates of the linear segment approximation were entered.

The resultant output is slightly distorted because the aspect ratio for the window and the viewport differ. The viewport aspect ratio defaulted to 1.52 (380 mm wide by 250 mm high) and the aspect ratio for the window is 355/130 or approximately 2.73. Thus, the projected image appears distorted.

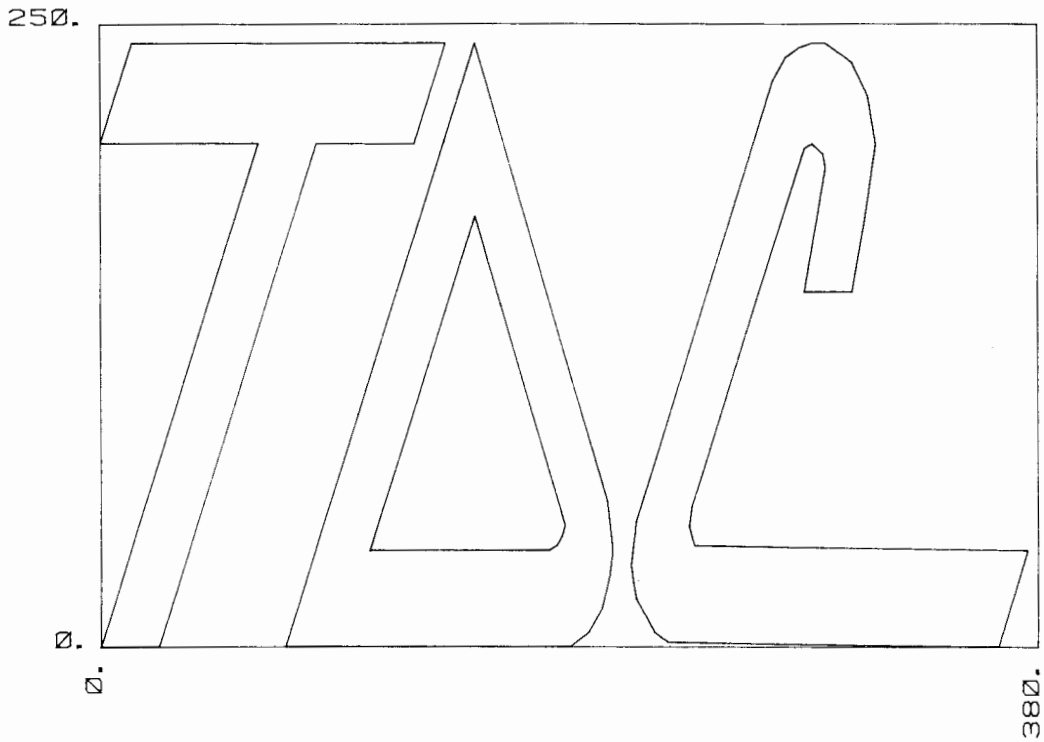


Figure 2.

2. Set a window to encompass the logo sketch; set viewport aspect ratio to correspond to window aspect ratio (SETAR); and default LIMIT and VIEWP parameters (comment out line 26 and 28).

```
CALL WINDW(IGCB,0.,355.,0.,130.)
CALL SETAR(IGCB,2.73)
```

The resultant output is an undistorted projection of the logo centered in the vertical direction and covering the total viewing surface in the horizontal direction (see figure 3).

The entire logo was projected, because the window was determined in the same manner as the previous example.

The projection was centered in the vertical direction and covered the total horizontal view surface because of the reconfiguration of the viewport that occurs when the SETAR routine is implemented. With the implementation of SETAR to 2.73, the reconfigured viewport corresponds to a rectangle with an aspect ratio of 2.73 being shrunk down until it just fits within the old viewport. When a rectangle with a 2.73 aspect ratio is shrunk to fit within a rectangle with a 1.52 aspect ratio, the horizontal dimension will be totally filled and the vertical dimension will have unused space. Therefore, as prescribed by the SETAR routine, the reconfigured viewport will be centered in the vertical direction and totally cover the horizontal view surface. When the image within the window is mapped to this reconfigured viewport it will project an image covering the horizontal direction and centered in the vertical direction.

The resultant image is undistorted because the aspect ratio of the viewport was designated as 2.73 by SETAR, and the aspect ratio for the window was also 2.73.

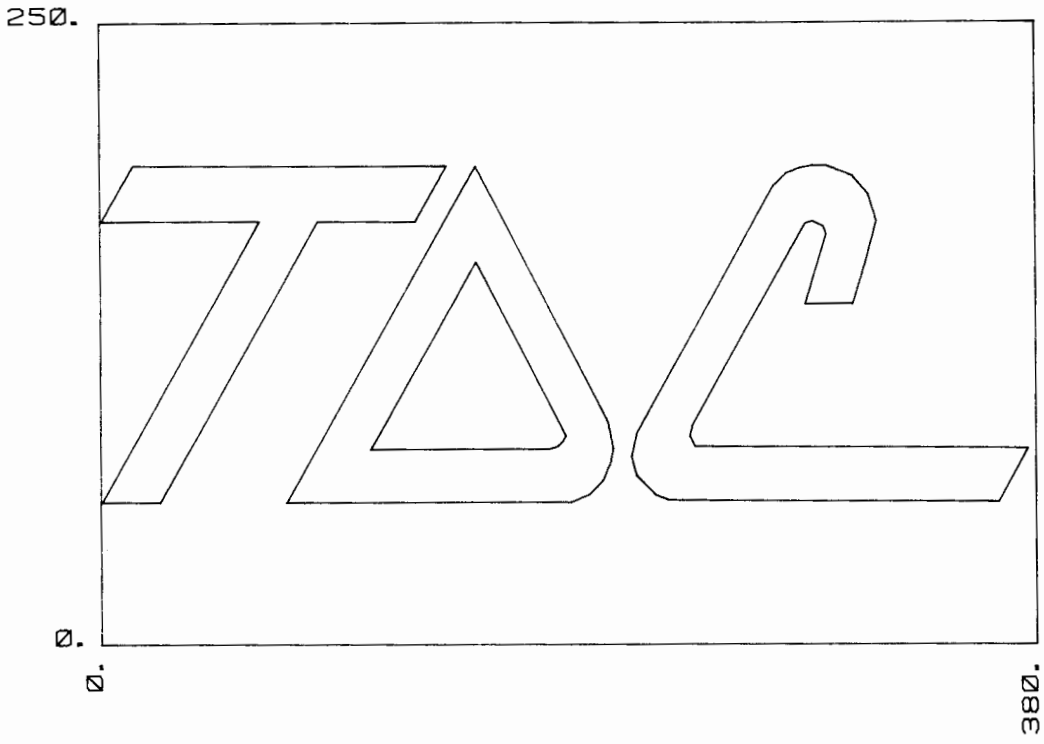


Figure 3.

3. Set a window to encompass the logo sketch; set the viewport aspect ratio to correspond to the window aspect ratio (SETAR); set the physical viewing surface (LIMIT) to correspond to an 8.5" by 11" viewgraph; default the VIEWP parameters (comment out line 28).

```
CALL LIMIT(IGCB,0.,215.,0.,280.)
CALL SETAR(IGCB,2.73)
CALL WINDW(IGCB,0.,355.,0.,130.)
```

The resultant output is an undistorted projection of the logo centered in the vertical direction and covering the width of a viewgraph (see figure 4).

The restriction of the projection to an 8.5" by 11" viewgraph placed on the lower-left portion of the device is a straightforward result of setting LIMIT. VIEWP is still defaulted to LIMIT and the other parameters are the same as for the previous example. Thus it follows that as the viewport is shrunk down to fit within LIMIT, it will fill the viewgraph in the horizontal direction and be centered in the vertical direction.

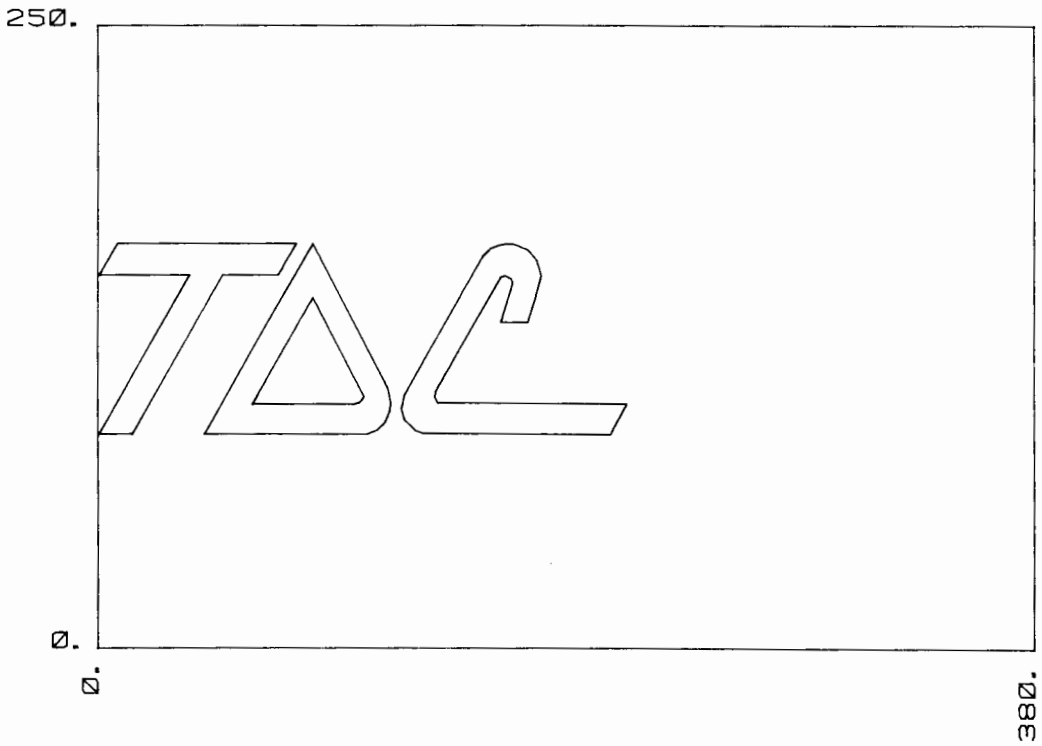


Figure 4.

4. Set a window to encompass the logo sketch; set the physical view surface (LIMIT) to correspond to an 8.5" by 11" viewgraph; set the viewport to five different locations (line 28); default SETAR (comment out line 27).

```
CALL LIMIT(IGCB,0.,215.,0.,280.)
CALL WINDW(IGCB,0.,355.,0.,130.)
CALL VIEWP(IGCB,6.,40.,6.,18.5)      Lower left  AR = 2.73
CALL VIEWP(IGCB,40.,94.,6.,25.8)     Lower right AR = 2.73
CALL VIEWP(IGCB,6.,70.,100.5,124.)   Upper right AR = 2.73
CALL VIEWP(IGCB,70.,94.,116.2,124.)  Upper right AR = 2.73
CALL VIEWP(IGCB,20.,80.,25.8,100.5)  Center      AR = 0.79
```

The resultant output is five projections of the logo within the 8.5" by 11" viewgraph area. The four corner projections are of various sizes all of which are undistorted. The center projection is distorted from that of the original sketch (see figure 5).

To understand the significance of the VIEWP parameters entered, the consequence of defaulting the parameters of the SETAR subroutine must be considered. Not calling SETAR defaults the viewport to the area delimited by LIMIT (the 215 mm by 280 mm viewgraph area). Since the viewport is a rectangle corresponding to the viewgraph area, it has an aspect ratio of 0.768 (215/280 = 0.768 = AR). Therefore, the height of the viewgraph corresponds to 131 units for the purpose of determining the vertical parameters for viewport positioning. (Referring to the explanation of VIEWP: If AR is less than 1, then the vertical view surface = 100/AR units = 100/.768 = 131 units). In a like manner, the width of the viewgraph corresponds to 100 units. (If AR is less than 1, then the horizontal length corresponds to 100 units.) Now it is of little consequence to calculate parameters for positioning desired viewports. But, if undistorted projections are desired, the viewports must be defined with an aspect ratio equal to that of the window (2.73). Subsequently, the corner viewports (calculated with an aspect ratio of 2.73) generated undistorted projections; while the center viewport calculated with an aspect ratio of 0.79 generated a distorted projection.

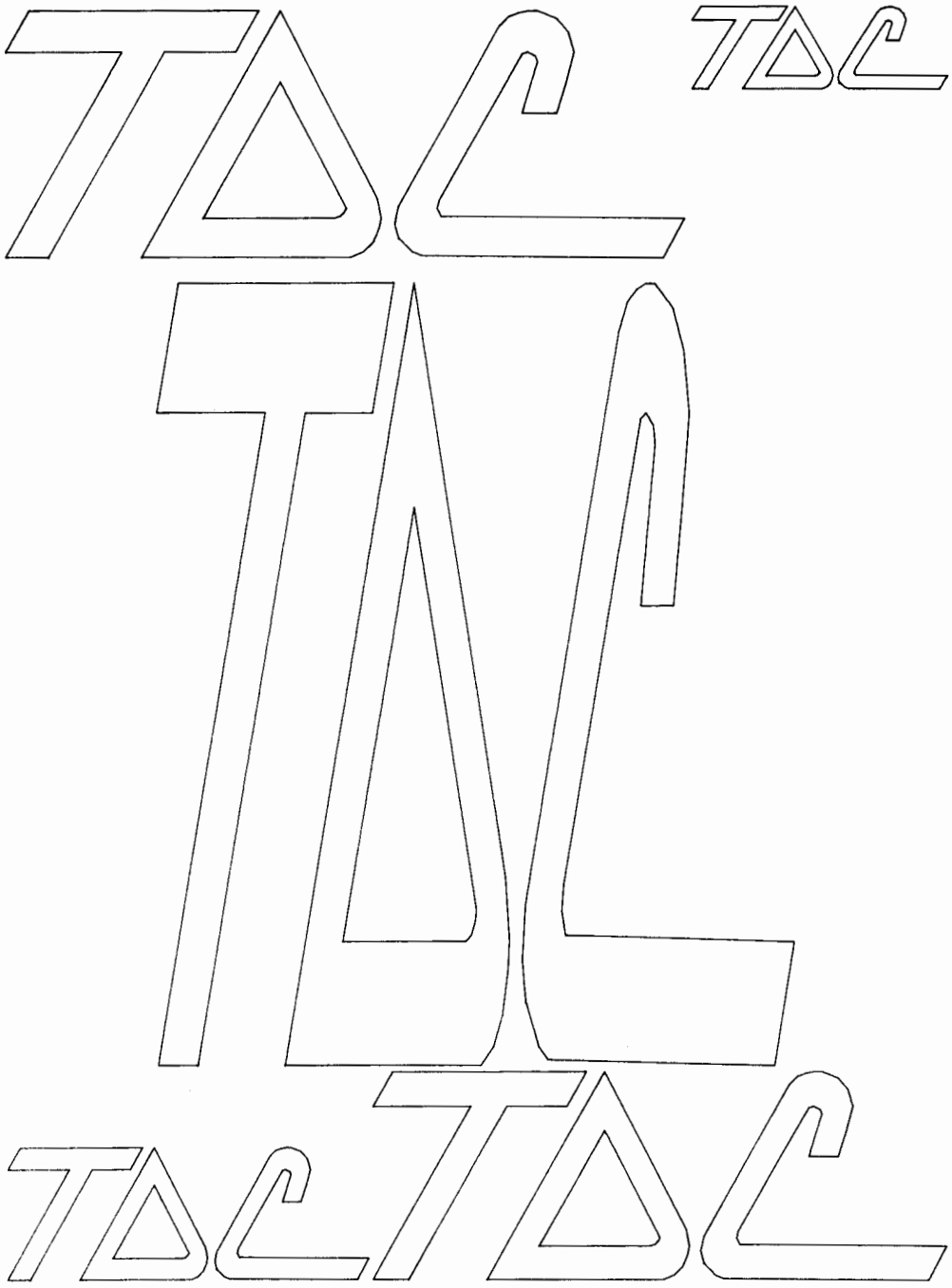


Figure 5.

SUMMARY

The reader should now be a "graphics expert". Not really. He may still not know a "logical view surface" from an illogical one. And, "normalized device coordinates" may seem entirely abnormal. What is hoped is that the reader now has a broad enough perspective of the graphics process that he can attack more detailed technical explanations and still keep the "forest in sight despite all the trees".

```
FTN4,L
      PROGRAM LOGO
C
      THIS PROGRAM DRAWS A LOGO "TDC"    (KATHLEEN SANDIFUR 10-79)
C
      DIMENSION IGCB(192),IBUF(10),XLUT(5),V(4),W(4),G(2),XL(4)
      EQUIVALENCE (LU,IBUF),(ID,IBUF(10))
C
C     GET ID AND LU
C
      LUT=LOGLU(I)
      WRITE(LUT,01)
   01 FORMAT("ENTER LU, ID:_")
      READ(LUT,*) LU,ID
C
C     INITIALIZE PLOTTER & DRAW FRAME
C
      CALL PLOTR(IGCB,ID,1,LU)
      CALL PEN(IGCB,2)
C
C     ********************************************************************
C     *  REFERENCES IN ARTICLE TO LIMIT, SETAR, VIEWP, WINDW             *
C     *            REFER TO THE FOLLOWING CALLS                           *
C     ********************************************************************
C
      CALL LIMIT(IGCB,XL(1),XL(2),XL(3),XL(4))
      CALL SETAR(IGCB,AR)
      CALL VIEWP(IGCB,V(1),V(2),V(3),V(4))
      CALL WINDW(IGCB,W(1),W(2),W(3),W(4))
C
C     ********* DRAW "T" ************************
C
      CALL MOVE(IGCB,22.,0.)
      CALL DRAW(IGCB,82.,105.)
      CALL DRAW(IGCB,119.,105.)
      CALL DRAW(IGCB,131.,126.)
      CALL DRAW(IGCB,12.,126.)
      CALL DRAW(IGCB,0.,105.)
      CALL DRAW(IGCB,60.,105.)
      CALL DRAW(IGCB,0.,0.)
      CALL DRAW(IGCB,22.,0.)
C
C     ********* DRAW "D" ************************
C
      CALL MOVE(IGCB,70.,0.)
      CALL DRAW(IGCB,178.,0.)
      CALL DRAW(IGCB,185.,3.)
      CALL DRAW(IGCB,190.,8.)
      CALL DRAW(IGCB,193.,15.)
      CALL DRAW(IGCB,194.,20.)
      CALL DRAW(IGCB,193.,25.)
      CALL DRAW(IGCB,192.,30.)
      CALL DRAW(IGCB,190.,34.)
      CALL DRAW(IGCB,142.,126.)
      CALL DRAW(IGCB,70.,0.)
```

```
C
C      ********* INNER "D"  START LEFT BOTTOM ****
C
       CALL MOVE(IGCB,102.,20.)
       CALL DRAW(IGCB,142.,90.)
       CALL DRAW(IGCB,175.,27.)
       CALL DRAW(IGCB,176.,25.)
       CALL DRAW(IGCB,175.,23.)
       CALL DRAW(IGCB,173.,21.)
       CALL DRAW(IGCB,170.,20.)
       CALL DRAW(IGCB,102.,20.)
C
C      ********* DRAW "C"  START LEFT BOTTOM ***********
C
       CALL MOVE(IGCB,203.,10.)
       CALL DRAW(IGCB,202.,13.)
       CALL DRAW(IGCB,201.,17.)
       CALL DRAW(IGCB,203.,26.)
       CALL DRAW(IGCB,255.,118.)
       CALL DRAW(IGCB,260.,123.)
       CALL DRAW(IGCB,265.,125.)
       CALL DRAW(IGCB,270.,126.)
       CALL DRAW(IGCB,275.,126.)
       CALL DRAW(IGCB,285.,122.)
       CALL DRAW(IGCB,291.,115.)
       CALL DRAW(IGCB,294.,105.)
       CALL DRAW(IGCB,290.,90.)
       CALL DRAW(IGCB,285.,74.)
C
C      ********* INNER "C"  START LEFT UPPER *****
C
       CALL DRAW(IGCB,267.,74.)
       CALL DRAW(IGCB,275.,100.)
       CALL DRAW(IGCB,274.,103.)
       CALL DRAW(IGCB,272.,104.)
       CALL DRAW(IGCB,270.,105.)
       CALL DRAW(IGCB,267.,104.)
       CALL DRAW(IGCB,224.,29.)
       CALL DRAW(IGCB,223.,25.)
       CALL DRAW(IGCB,224.,23.)
       CALL DRAW(IGCB,225.,21.)
       CALL DRAW(IGCB,351.,20.)
       CALL DRAW(IGCB,340.,0.)
       CALL DRAW(IGCB,215.,1.)
       CALL DRAW(IGCB,210.,3.)
       CALL DRAW(IGCB,203.,10.)
  9999 CALL PEN(IGCB,0)
       CALL PLOTR(IGCB,ID,0)
       STOP
       END
```

GENERATING RTE FOR PLEASURE AND PERFORMANCE

Shulom Kurtz/HP Englewood, Colorado

INTRODUCTION

As a Hewlett-Packard System Engineer working in the field with our I000 system users, I have found increasing evidence that system generation, in many cases, is just a matter of getting enough pieces "pasted together" in an answer file to make it work. Once that has been accomplished there seems to be little, if any, effort devoted to reviewing the system listings furnished by the generator, with the intent of optimizing the system; this leads, inevitably, to less than optimum performance, as well as to over utilization of certain limited system resources.

Let's examine some of the key factors involved in these challenges and work out a means of meeting the opportunities presented to us as users interested in getting MORE for LESS: MORE PERFORMANCE for LESS SYSTEM IMPACT.

USE ALL AVAILABLE MICROCODE

In terms of performance, the user should know explicitly what the hardware level of the system is. With the advent of the 21-MX series of processors, more and more of the instructions and subroutines are reduced to microcode. The section of system generation concerned with replacement of entry points builds a table used by both the on-line generator and the on-line loader, to substitute the faster microcode equivalents for software routines in all memory-image programs.

Do you know that your list of RPs is complete? A general rule of thumb at HP is that instructions are not microcoded unless a performance improvement of 10-to-1 or better can be anticipated. If your operating system has not been generated with all RPs appropriate to the hardware/firmware configuration, you are not taking full advantage of the speed available to you.

If you want more throughput in processor-bound routines, a quick check of your current system generation against the microcode available in your processor may be quite revealing. (Remember that the list of RPs in the grandfather disc answer file is absolutely minimal. The grandfather operating system will therefore, run on almost any hardware configuration.)

If you are unsure whether certain groups of microcode instructions are available on your machine, you can quickly code an assembly language module which consists of RPL instructions of the form

```
MNEM RPL xxxxxxB
```

where MNEM is the mnemonic for the instruction, and xxxxxx is the octal microcode address. Be sure that each of the mnemonics is also included in an ENT statement at the top of the code.

After this module has been assembled, present it to the loader before relocating any program modules; then load your key test program, with timing instructions included, and run it. If the program aborts as a result of either memory-protect (MP) violations or dynamic mapping system (DMS) errors, you have a pretty good clue that one or more of the microcode instructions does not exist on your machine. A look at the information presented by the operating system when the program has abnormally ended (ABEND) will tell you which instruction does not exist in your machine's microcode.

On the other hand, if the program completes normally, OF it, reload it withOUT the RPL module, and run it again, to compare the two timings. (Suggestion: to get a better indication of effectiveness, make your test program relatively short in terms of code and loop through it many times; 10,000 times is a good number.) You may be in for a pleasant surprise and, if so, be sure to include the microcode RPs in the answer file for your next system generation.

While we're on the topic of microcode, remember that HP encourages the development of user microcode for frequently used routines that appear to be processor bound. The writable control store (WCS) can be dynamically loaded with your routines as needed. After thorough testing, you can even have ROMs burned with permanent versions of that microcode, to be installed on the user control store (UCS) printed circuit board. Then your subroutines can have their microcode addresses RP'd during system generation to assure that the loader will automatically make the appropriate linkages. Your HP sales representative will be happy to give you more information about the WCS, UCS, and the microprogramming software subsystems. And yes, the Systems Engineering Organization has a customer course available for training you in the use of these valuable tools.

OPTIMIZE DRIVER RELOCATION FOR MORE MEMORY

Starting with RTE-IV and continuing in RTE-IVB, with certain exceptions, most drivers are located in memory-resident driver partitions within the operating system. Only one such driver partition is mapped into your user partition at any moment in time. As contrasted with earlier RTE systems, this means that your maximum program size may be increased considerably, since those earlier systems mapped all drivers into each user area. On the other hand, if you can reduce the number of driver partitions by only one, that gives you two additional pages of memory available for SAM, system common, or even user partitions!

As you read the manuals on system generation, you will find that the generator attempts to optimize driver partition occupancy. A look at the generation map will probably show that a rearrangement of the drivers might eliminate one partition. The conclusion, therefore, is obvious: the optimization algorithm in the generator is not very sophisticated.

After you have generated your system, look at the loading of the drivers, keeping in mind that the driver for the system disc must always be loaded in driver partition 1. Assuming that you find a fair amount of unused memory at the end of each partition, a little time spent computing the actual driver sizes, and rearranging the drivers to see if you can get more efficient use of each partition, may result in reducing the number of partitions by one. After you determine your best sequencing of partition-resident drivers, be sure that your answer file for sytem generation relocates the drivers in that exact sequence.

REDUCE THE NUMBER OF TRACKS YOUR SYSTEM REQUIRES

Libraries relocated during system generation are placed in the system area on LU2. Some of those libraries are only needed for loading HP provided utilities or language processors. By eliminating these libraries at generation time, you also require that the programs needing them be loaded on-line after running SWTCH. That's not all bad: you can save from one to two hours of generation time by doing that!

Before going on to describe the technique, let me explain this last statement. A major function of the on-line generator is to act as a gigantic loader, relocating all programs and subroutines presented to it. A loader must create tables of external references for each relocatable module, and attempt to resolve those references with entry-point addresses found in all the relocatable modules given to it. Search time through these tables goes up EXPONENTIALLY in proportion to the number of symbols involved. Again, the conclusion is obvious: cut down the number of symbols by reducing the number of programs, and you can save generation time. On-line loading takes far less time because the loader has only one relocatable module to process at a time.

The libraries I recommend eliminating from generation are:

- The compiler library (%CLIB) which is required for loading the FORTRAN-IV compiler, the Assembler, the Cross-Reference Table Generator, the Microprogramming Assembler, and the Basic Interpreter.
- The disc back-up utility libraries for loading the system back-up utility programs (%DBKLB, $DSCLB, and $DKULB depending on the version of software you are using).
- The utility library (%UTLIB) in RTE-IVB required for loading WRITT, READT, COMPL, and CLOAD.

Remember, all of the programs listed above will have to be loaded on-line after running SWTCH. Therefore, you will need to copy the relocatable modules for the programs and libraries not generated into your system, put them on your disc after switching in your new system, and then load them. This means you must increase the number of blank long and short I.D. segments during system generation to provide for these programs, if you are going to load them permanently. Specifically, you should add six short I.D. segments for the FORTRAN compiler, 5 for the assembler, and 8 for the BASIC interpreter so that you can, during boot-up, restore (RP) the segments in your WELCOM file. After each program is loaded — if it is not loaded permanently — you will want to save (SP) the programs as type 6 files, and then OF them to free the I.D. segments and the system tracks used during the loading process. Also, be sure that if a program is segmented, you SP the segments on the same system cartridge as the main, and OF each of the segments in a similar fashion.

Depending on your operating system's loader (we have had several, each increasingly more sophisticated), the technique of loading these programs on-line without their associated libraries in the system library will vary. Under RTE-IV and RTE-IVB, you simply declare " LI,%xxxxx " as one of the loader commands before starting relocation, and the module named will be searched along with the system libraries. You can declare up to 10 such private libraries during the process of any load.

SHORTEN LOADING TIME FOR ALL OF YOUR ON-LINE LOADS

The generator sets up system libraries in the order which you relocate them. The implication here is that you should sequence your libraries so that the most often called HP subroutines are in the first library, etc. It is not necessary to go into extensive analysis of this approach, since the libraries are in modules which are internally already presequenced by HP. Hence, unless you want to work at breaking those modules apart and evaluating usage of each subroutine within the libraries, the following order is probably close to being optimal: %RLIB1, %RLIB2, %RLIB3, %FF4.N, %4SYLB (or the comparable system library for systems preceding RTE-IV), %SMON1 and %SMON2 (for RTE-IVB), %BMPG3 (the FMP library), the BSM library if you're including the spooling subsystem, and then any others you may require including user written subroutines you wish to make available for all programs to access.

Obviously, there is always the opportunity for you to make an independent determination that certain libraries late in the list above should be moved ahead. This will be dependent on your own evaluation of how programmers on your system will use the library routines.

EVALUATE HOW WELL SYSTEM RESOURCES ARE USED

As noted above, by eliminating certain lesser used library modules from a system generation, you can save tracks occupied by the system on LU2. Since the system track pool is important for use in program swapping, EDITR work space, on-line loader temporary and/or permanent loads, as well as for use on request by user programs, and, since LU2 is limited to a total of 256 tracks, any conservation of this resource is important. It becomes even more significant if you choose not to generate LU3 as the system auxiliary disc.

In general, I recommend that the auxiliary subchannel (LU3) be established for any RTE system. It extends the otherwise limited system resources and can, at the same time, be used for any other purposes you may desire.

There is a trade-off to be considered in this respect between the recommendations made to reduce the number of system tracks and the converse, i.e., in the area of using the generator to make permanent loads of programs. Specifically, the generator allocates disc space on a block requirement basis (physical sector of 128 words), whereas the on-line loader allocates space on a per-track basis. If a permanently loaded program requires considerably less than a full track, the unused portion of the track is wasted. Of the programs included in the list for loading after running SWTCH, only VERFY and XREF need existing I.D. segments before running either the utility programs or the assembler. Rather than use a permanent load, thus tying up tracks on LU2, RP these two programs from type 6 files in the WELCOM file at boot-up, and the I.D. segments will be there without losing any more LU2 tracks than absolutely necessary.

System available memory is more heavily used under Session Monitor than in MTM or single console systems. A session control block is allocated for each user logged on the system. If extensive automatic buffering of I/O is also required, large amounts of class I/O are used, or frequent calls are made to REIO (as opposed to EXEC I/O), serious performance degradation may be experienced unless there is enough SAM to meet your needs!

Contrary to Table 3-6 in the RTE-IVB System Manager's Manual, the ACCTS program requires 17 pages (not 16). If you have large system drivers and/or large Table Area II entries, you may be dismayed at the generation listing of maximum program size for type 3 programs (such as ACCTS prior to 2001). Since there is little to be done about driver size, the only recommendation is to work on cutting down some of the table area II entries. Reduce, where feasible the number of blank long and short I.D. segments, the number of LU mappings (required only for spooling operations), etc.

Memory allocations during system generation may cause just a "slight" break across page boundaries. This is particularly true with system common, SSGA, etc. Watch relative addresses closely on your listing of the generation. The allocation of real time system common is totally under system manager's control; background common is allocated by default to the next page boundary. A review of system common usage may indicate need for a change in the mode of usage — but be careful! If you have existing programs, this may force a lot of reprogramming effort! Only you can determine the merits of the trade-off between the size of the common areas and SAM, and the size of the system.

This article represents a summary of some of the major points I have been able to pin down for improving system generations. It is not intended to be either comprehensive or absolute. Rather, the purpose of this article is to stimulate thought and analysis on the reader's part. If you have additional ideas, I join the editor of "The Communicator" in urging you to share them with the rest of the 1000 users through these pages — either in an article or as a "short shot" in "The Bit Bucket."

[Editor's note: Shulom's article ties together several articles that have appeared in the Communicator recently. Bob Niland's "Links 1000" series that has had parts appearing in the Communicator since Volume III Issue 6, gives an excellent and detailed description of the use of firmware microcode. Ed Gillis' article in Volume IV Issue 2 titled "List of Drivers and Their Sizes" can help the reader optimize the use of partitions when relocating drivers in a generation.]

HP SUBROUTINE LINKAGE CONVENTIONS

Robert Niland/HP Lexington, Ma.

[Editor's note: This article is the fourth in a series of six contributed by Bob. The series began in Volume III issue 6.]

5-1. THE .ENTR LINKING ROUTINE

In Chapter 4 we saw that placing parameters in common has significant advantages over the methods of Chapter 3 (which are generally available only to the assembly programmer anyway). However, even the most flexible COMMON implementation, NAMED COMMON/SSGA has serious pitfalls and should not be used for routine parameter passing.

Ideally we should like to have a scheme whereby we can pass to a subroutine only those parameters which it needs, and one such that the subroutine need not know anything about the parameters except what data type they are.

It is of course no secret that all implementations of Fortran have this capability. Examples are:

```
CALL SUBRU (PARAM1,PARAM2)
RESULT = FUNCT (PARAM3,PARAM4)
```

Generating the appropriate binary code in the calling program and in the subroutine is handled transparently by high level language processors. This chapter will describe the HP1000 implementation.

The centerpiece of the scheme is a utility subroutine named .ENTR, which is supplied in the DOS/RTE Relocatable Library. On F-series computers, 2100A/S, M, and E-series computers with Fast Fortran Processors, this routine is provided in firmware. The link from software to firmware is described in section 3-5 of this series. The privileged version of .ENTR (.ENTP) is described in chapter 6.

.ENTR capabilities:

- Calling language: ANY
- Called language: ASMB,FTN,FTN4
- Type of passed parameters: ANY, although the subroutine cannot programmatically determine what type each is, and therefore this information must be coded for each parameter.
- Number of passed parameters: NO LIMIT
  Limited only by program address space. One word in the caller and one word in the subroutine are consumed for each parameter. A parameter count of zero is also legal, although some of the Fortran compiliers will insist on a dummy variable.
- Agreement in order: Required.
- Agreement in number: Not required.
  .ENTR will only attempt to pass as many as were declared by the caller, but in no event more than the subroutine is expecting. Parameter defaulting is discussed in section 5-2.

In a nutshell, .ENTR works by passing addresses of parameters. It is important to understand this. .ENTR does not pass the actual parameters, only their addresses. This means that .ENTR only supports call-by-reference, and does not directly support call-by-value. Call-by-value on the HP1000 essentially requires the creation of a temporary variable containing the value to be passed, and then passing that temporary variable by reference.

In normal use, .ENTR is only invoked by the subroutine, not the calling program or subroutine. The caller's code need only conform to .ENTR's expectations. The form of a .ENTR-compatible code in the caller is shown in assembler:

```
EXT SUBRU                EXT SUBRU        (Unless SUBRU is in caller)
...                      ...
JSB SUBRU                JSB SUBRU        Call subroutine
DEF *+n+1      -or-      DEF ]RET         Define return point.
DEF PRAM1                DEF PRAM1        Define 1st parameter address.
DEF PRAM2                DEF PRAM2        Define 2nd parameter address.
DEF @PRM3,I              DEF @PRM3,I      Addresses may be indirect.
...                      ...              More parameter addresses.
DEF PRAMn                DEF PRAMn        Last parameter address.
...              ]RET    EQU *            Return point.
```

There are three or four steps:

1. JSB to the subroutine's entry point in the normal fashion.

2. Follow the JSB with the direct or indirect address of the next executable instruction in the caller. This is normally the location following the last parameter address. This DEF <return>may be relative (* +n+1, where * =current value of P, n =number of parameters), or it may be symbolic (DEF]RET, where]RET is the label of the next instruction or a dummy EQU * line). The symbolic form is preferred as it will always result in a proper subroutine return, even if the original programmer couldn't count (n was wrong) or the number of parameters was subsequently changed.

3. DEFine the addresses of (direct) or links to (indirect) each parameter to be passed.

4. If the return point is symbolic, don't forget to place the symbol in the next location to be executed.

The Fortran equivalent of the above call is:

```
CALLSUBRU (PRAM1,PRAM2,PRAM3,....,PRAMn)
```

Detailed discussion of high-level language implementations will be deferred until chapter 7.

The perceptive reader has probably noticed that the form of the assembly level call shown above is only slightly different from the instruction emulation examples of chapter 3. The call is a JSB followed by some number of DEF's. The important distinction is that in the case of a .ENTR compatible call, one of the DEF's is to the return point. This allows .ENTR, when invoked by the called subroutine, to determine both the actual number of parameters and the final return point.

Given the informative format required for a .ENTR compatible call, a clever assembly programmer could write subroutines which "reach" back through their entry points and figure out how many parameters there are, what their ultimate true addresses are, and where the return point is. And this is sometimes done, especially in cases where one and only one parameter is passed. But, it makes far more sense to write a utility routine to do it, and that's all .ENTR is.

.ENTR is called by the subroutine to link the subroutine to its caller. .ENTR performs three functions:

1. Passes the parameter addresses to the subroutine.

2. Insures that they are direct addresses.

3. Adjusts the return point. This step is necessary, for if the subroutine attempted to return by executing a JMP SUBRU,I without adjustment, it would actually return to DEF * +n +1 or DEF]RET and attempt to execute the DEF and the parameter addresses as if they were instructions. .ENTR resets the value in the entry point to the direct address * +n +1 or]RET so that the subroutine may perform an ordinary return.

The calling sequence in the subroutine is:

```
        ...
        ENT SUBRU          (unless SUBRU is in the caller).
        ...
@PRM1   MPX                Location for address of parameter #1 (link 1)
@PRM2   MPX                Location for address of parameter #2.
@PRM3   MPX                Location for address of parameter #3.
        ...                etc.
@PRMn   MPX                Location for address of parameter #n.
SUBRU   MPX                Entry point/raw & corrected return address.
        JSB .ENTR          Call utility routine .ENTR.
        DEF @PRM1          Address of the start of SUBRU's address list.
        ...                SUBRU's application code begins.
```

The rules are:

1. Immediately above the entry point, reserve one memory location for each expected parameter. This is done with MPX pseudo-instructions in the example, but could have been done with any other ASMB instruction which consumes one memory word, such as NOP, BSS 1, DEF DEFLT, etc.

2. The locations for these links must be contiguous and in the same order as passed. Comments and zero length pseudo-instructions may appear between each link.

3. There must be no memory consuming instructions between the last link location and the subroutine entry point. .ENTR computes how many parameters are expected by subtracting the address of the entry point from DEF @PRM1.

4. There must be no memory consuming instructions between the entry point and the JSB .ENTR. .ENTR makes this assumption, and looks in the location prior to your JSB .ENTR to get the address of the DEF *+n+1 in the original caller.

5. The address of your link-list must be defined in the location following the JSB .ENTR. .ENTR needs this for step 3.

When .ENTR completes, you will have the following:

1. The direct addresses of passed parameters 1..n in link locations @PRM1....@PRMn. If k parameters were passed, and k is greater than n, the addresses of parameters n+1 to k in the caller are ignored. If k is less than n, links @PRMk+1 to @PRMn are ignored. Links @PRMk+1 to @PRMn will contain whatever was previously in them, whether stored there by the assembler, a prior call, or by SUBRU's own activities.

2. The address in the entry point (SUBRU), will be reset to the direct address specified in the location just after the caller's JSB SUBRU. It will be address *+n+1, or, for the symbolic method, the address of the location following the last parameter declared in the caller, whether passed or ignored by .ENTR.

3. Execution will resume at (P will be set to) the location following the DEF @PRM1, with the A and B registers set to the values specified in the Library or firmware manuals, E and O undefined, and X an Y (if present) unchanged.

In the body of the subroutine code, consider:

1. All your accesses to passed parameters are with reference to the first word, as all you have is its address. For simple variables, no address offset calculations may be needed. For example PRAM1 = PRAM2 + PRAM3, could be coded as:

```
LDA @PRM2,I       Get parameter 2.
ADA @PRM3,I       Add to parameter 3.
STA @PRM1,I       Return result.
```

   For vectors and arrays, address computation will be needed. PRAM1(35) = FLOAT (PRAM3) might be:

```
LDA @PRM1         Get address of PRAM1(01)
ADA =D34          Compute address of PRAM1(35)
STA @TEMP         Save temporarily.
LDA @PRM3,I       Get PRAM3
FLT               Float it.
DST @TEMP,I       Store in PRAM(35)
```

   For arrays of two or more dimensions, use ..MAP.

2. The return to the caller is accomplished with a normal

```
JMP SUBRU,I
```

   jump through the entry point. .ENTR has assured us that this is where we want to go. Note that although the working registers A..O are modified or lost during subroutine entry, they are not disturbed during return, and so may be set to any desired value. This feature is used by some high-level language conventions.

Refer to the listing of subroutine TBLE in Appendix D for an example of a simple .ENTR call.

5-2. DEFAULTING PARAMETERS WITH .ENTR

As we have seen in the list of .ENTR's capabilities, the caller may pass fewer parameters than the subroutine is expecting. For example in RTE, an OPEN call might take the forms:

```
IMPLICIT INTEGER (A-Z)
...
CALL OPEN (DCB,ERROR,NAME)
CALL OPEN (DCB,ERROR,NAME,OPTION)
CALL OPEN (DCB,ERROR,NAME,OPTION,SECURE)
CALL OPEN (DCB,ERROR,NAME,OPTION,SECURE,CRLU)
CALL OPEN (DCB,ERROR,NAME,OPTION,SECURE,CRLU,DCBSIZ)
...
```

In each case the default value for the omitted parameter(s) is supplied by the OPEN subroutine. These are:

```
CALL OPEN (<dcb>,<error>,<name>,0,0,0,128)
```

This is a useful technique to use when a subroutine has an extensive parameter list, organized from most to least significant, and where the less significant parameters are frequently set to some known value. The subroutine (which must be written in assembly language) can be coded to "know" what the defaults are. Of course, only trailing parameters may be defaulted. For good reason, the language processors will not permit us to use:

```
CALL OPEN (DCB,ERROR,NAME,,SECURE,,DCBSIZ)
```

In a nutshell, the subroutine allows the last (several) parameters to default by taking advantage of the fact that .ENTR will not modify the existing contents of link locations associated with parameters not supplied by the caller. Two steps are required to make this work:

1. The contents of the link location must have been preset to a useful value prior to calling .ENTR in the subroutine.

2. Since the contents of the link location may be modified by a call which does not default parameters, the link must be reset to the default value prior to subroutine exit, so that subsequent calls using the default feature will get the true default values, and not the explicit value set up by some prior call.

Before addressing the issue of how to preset/reset the links, let's consider the contents of the link itself. If it were preset/reset to MPX (102000B) or NOP (000000B), our subroutine could easily tell when a default had occurred, as .ENTR would never set a link to an indirect address (102000B = 2000B,I) or to zero (the A register), but this would require excess code. In the NOP case for example:

```
...             Code subsequent to JSB .ENTR
LDA DEFLT       Get default for parameter 3.
LDB @PRM3       Get contents of link three.
SZB             Skip if NOP (PRAM3 was defaulted).
LDA B,I         Wasn't defaulted, get PRAM3.
...             Now PRAM3 or its default is in A.
```

It would be far more efficient to preset/reset the link to what the main code of the subroutine normally expects, i.e. an address.

Suppose we wish to write a subroutine which can force all words (character pairs) in a buffer to be members of the ASCII set, or some subset of it, and the calling sequence is:

```
CALL ASCII (<buffer>, [,<length> [,<set>]])
```

Where: **<buffer>** is the character string to be processed, and can not be defaulted.

<length> is the size of <buffer> in +words or −bytes, and has a default value of +1

<set> is the identity of the ASCII set/subset desired, and is 64, 96 or 128, with a default of 128.

The source entry code for ASCII might look like this:

```
        ...
@BUFR   MPX             Target buffer (word address).
@SIZE   DEF D0001       BUFR size in +words, -bytes. Default=+1
@SET    DEF D0128       ASCII set desired: 64, 96, default=128.
        JSB .ENTR       Set up links, adjust return address.
        DEF @BUFR       Parameters start at @BUFR
        ...
```

Where D0001 and D0128 are the symbolic locations of the default decimal values in ASCII's own code, i.e.,

```
D0001 DEC 00001
D0128 DEC 00128
```

5-3. PRESERVING REGISTERS WITH .ENTR

In the summary about .ENTR completion we noted that the states of the working registers (A..O) in the calling program are not preserved when the subroutine calls .ENTR to resolve links. If the purpose of the subroutine requires information in those registers, at first glance it appears that we may not be able to use .ENTR. This is not the case, but before the solution becomes apparent, we need to examine how .ENTR works in a little more detail than was presented in section 5-1.

Let's re-state the subroutine's calling rules from page 5-3:

1. Above the apparent entry point, reserve one word per link.

2. Links must be contiguous and in agreement with the caller's order.

3. There may be memory locations between the last link and the apparent entry point, but you must be certain that no caller will attempt to pass more parameters than expected, or these locations will be over-written.

4. The memory location just prior to the JSB .ENTR (the apparent entry point) must contain the "raw" return address of the subroutine's caller. .ENTR does not care how that address gets placed before the JSB. Nor does .ENTR care if that location is the actual entry point to the subroutine.

5. The location just following the JSB .ENTR must contain the direct or indirect address of the link area of rule 1.

With this information we can now construct a way to "fake-out" .ENTR. Let's suppose we want to write a subroutine which can return the current state of A,B,E,O,X and Y to a Fortran caller. It's calling sequence will be:

```
IMPLICIT INTEGER (A-Z)
...
CALL STATE (A [,B [,E [,O [,X [,Y]]]]])
```

The overall flow of such a routine will have to be:

1. <deferred>
2. Save the working set of registers.
3. <deferred>
4. Link to caller with .ENTR.
5. Pass the working set of registers back.
6. Reset default links & return.

Inside the routine ASCII, its code can access the buffer size and set indicator with the ordinary methods:

```
        ...
        LDA @SIZE,I       Get actual/defaulted size.
        ...
        LDB @SET,I        Get set number.
        ...
```

ASCII's code does not have to test the links to see if they have been defaulted. It automatically gets the explicit (passed) or local (defaulted) parameter values.

As in any implementation of parameter defaulting, however, we do have to reset the links on exit. For, if the 3rd call to ASCII was:

```
CALL ASCII (STRING,-33,64)
```

and we did not reset the links, the next defaulting call such as:

```
CALL ASCII (IWORD)
```

would not assume a 1 word (2 byte) buffer and a 128 character ascii set, but would still contain addresses indicating a 33 byte buffer and a 64 character ascii set. The addresses placed in these locations after the previous call would still be intact. In this example the code needed to reset the addresses would look like the following:

```
        ...
        LDA @D001         Get address of SIZE default.
        STA @SIZE         Reset @SIZE to default link value.
        ...
        LDA @D128         Get address of SET default constant.
        STA @SET          Reset @SET to default link value.
        ...
        JMP ASCII,I       Return external.
        ...
@D001   DEF D0001         Address of default value 0001.
@D128   DEF D0128         Address of default value 0128.
        ...
```

So to summarize the (suggested) rules of parameter defaulting:

1. Preset the link locations with the assembler, using pseudo-instructions, preferably a DEF <default>.

2. Write the subroutine's application code as if the parameters were not able to be defaulted.

3. Reset the link locations on exit via:

   a. Loading the address of the <default> and store it in the link.

   b. Caution: If the subroutine's code (step 3) is performing address computations on the defaultable links, or is using their contents as the operand of any operation which requires a direct address (such as a CMW instruction), you cannot rely on the DEF pseudo-instruction to generate that direct address. HP will not even guarantee that a DEF <symbl>+0 will generate a direct address. You must use a resolving routine, such as .DRCT to insure a direct address. Some of the sample routines in the appendix use the author's own routines .DAA, .DAB, .ALA, and .BLA to perform this resolution.

The entry into the subroutine is the key, and this is how it is done:

```
        EXT .ENTR
        ...
STATE   MPX                     Actual entry point. Raw return address.
*                               *
        DST A..B.               Save A and B locally.
        CLA,SEZ                 Presume E is clear, skip if true.
        INA                       E Isn't clear, set A=1.
        STA E..                 Save locally.
        CLA                     Presume O clear.
        SOC                     Skip if true.
        INA                       O Isn't clear, set A=1.
        STA O..                 Save locally.
*                               *
        LDA STATE               Get raw return address.
        STA PSEUD               Stuff in pseudo-entry point.
        JMP PSEUD+1             Simulate caller's JSB.
*                               *
@A      MPX                     Link for [A]
@B      DEF NULL                Link for B, or local bit-bucket.
@E      DEF NULL                Link for E, or local bit-bucket.
@O      DEF NULL                Link for O, or local bit-bucket.
@X      DEF NULL                Link for X, or local bit-bucket.
@Y      DEF NULL                Link for Y, or local bit-bucket.
PSEUD   MPX                     Pseudo-entry point/return address.
        JSB .ENTR               Resolve links, adjust PSEUD.
        DEF @A                  Start with 1st link.
        ...
        ...                     Return A.. to Y.. through links.
        ...                     Reset defaulting links.
        ...
        JMP PSEUD,I             Return external.
```

Now steps 1 and 3 whose explanations were deferred above can be added to the recipe:

1. Code the subroutine with two entry points, a real one which has the EXTernal name by which the routine is known, and a pseudo entry point which will be used by .ENTR to set up the links (and the true return address).

2. Same as before.

3. Copy the raw return address from the real to the pseudo entry point, and jump to the JSB .ENTR, just as though the caller had JSB'd to it.

4. Same as before.

5. Same as before.

The complete listing for STATE is in Appendix E.

5-4. MULTIPLE ENTRY POINTS WITH .ENTR

In an assembly language subroutine, more than one ENT pseudo-instruction may be used, i.e. portions of the same code may be called by different names. Let's see a circumstance in which this feature could be useful.

Let us suppose that we wish to write a subroutine which can set all words in an FMP file to any desired value. It might have a calling sequence of:

```
CALL ERASE (DCB,ERROR,BUFFER,LENGTH [,EOFLAG [,MASK]]
```

Where: **EOFLAG** = positive number of records
−1 = erase for computed (LOCF) file size.
−012 = erase until −012 eof error (default).
MASK = word to write into all file words (default =0).

With the advent of RTE-IVB, the routine above will suffice for ordinary sector-sized files, but we will also need a version for chunk-sized files:

```
CALL ERASF (DCB,ERROR,BUFFER,DLENTH [,DEOFLG [,DMASK]])
```

Where the parameters are identical except that the last three parameters must be passed as double integer. (DEOFLG and DMASK are still optional parameters.)

We could write separate subroutines for the two types of files, but since the logical code for these two routines would be identical, and their internal FMP calls (e.g. WRITF,EWRIT) would differ in name only, it might be advantageous to have only one routine with two names.

Using what we have learned about .ENTR to this point, this is a fairly straightforward task. We simply use the "saving registers" technique of section 5-3 twice. However, in this case we are not saving registers. What we are going to do is save a value, and set a flag which will tell the main body of the subroutine which entry point we came in through.

For example:

```
        ...
        ENT ERASE,ERASF *Do not declare PSEUD to be an ENTry.
        ...
ERASE   MPX             Real entry point for sector-files.
        LDA ascSE       Get "SEctor-file" flag value.
        STA file?       Set entry-point identifier variable.
        LDA @M012       Get address of EOFLAG default.
        STA @EOF        Preset link which can default.
        LDA @0000       Get address of MASK default.
        STA @MASK       Preset default.
        LDA ERASE       Get our raw return address.
        STA ERASx       Store in pseudo-entry point.
        JMP ERASx+1     Make a pseudo-JSB entry.
*                       *
ERASF   MPX             Real entry point for chunk-files.
        LDA ascCH       Get "CHunk-file" flag value.
        STA file?       Set entry-point identifier variable.
        LDA @I012       Get address of DEOFLG default.
        STA @EOF        Preset link which can default.
        LDA @0000       Get address of DMASK default.
        STA @MASK       Preset default.
        LDA ERASF       Get our raw return address.
        STA ERASx       Store in pseudo-entry point.
        JMP ERASx+1     Make a pseudo-JSB entry.
```

```
*                          *
@DCB  MPX                  file Data Control Block.
@ERROR MPX                 returned integer ERROR.
@BUFR MPX                  buffer ERASx needs for reads & writes.
@SIZE MPX                  size of that buffer in integer words.
@EOF  MPX                  "erase until" flag. default= -012 (eof)
@MASK MPX                  value to write into file. default=0.
*                          *
ERASx MPX                  Pseudo-entry/real return point.
      JSB .ENTR            Get links, adjust return address.
      DEF @DCB             Start at @DCB.
      ...
      ...                  application code.
      ...
      JMP ERASx,I          Return to the external routine through the
      ...                  pseudo-entry point.
I0000 DEC 00000,00000       Integer & double-word integer zero.
I0012 OCT 177777            1st word of double-integer -12.
M0012 DEC -00012            2nd word of DBL -12, entire INT -12.
@M012 DEF M0012             Address of integer -12.
@I012 DEF I0012             Address of double-integer -12.
@0000 DEF I0000             Address of either zero.
```

Note that another concept has been introduced here, pre-setting the default links at run-time, but prior to calling .ENTR. This action is necessary here because two different binary number formats are possible in the parameters that can default.

Within the applications code, we can determine what type of file we have been asked to erase by examining the variable "file?". For example, at the point where the erase buffer is written:

```
      ...
      LDB file?            Get the file type.
      LDA @WRTF            Assume a WRITF will be required.
      CPB ascSE            Skip if file?="SE" and use WRITF
      LDA @EWRT               if file?="CH", use EWRIT instead.
      ...
      JSB A,I              Call WRITF or EWRIT.
      DEF ]WR              Return point.
      DEF @DCB,I           Pass caller's DCB.
      DEF @ERROR,I         and caller's ERROR.
      DEF @BUFR,I          point at caller's buffer.
      DEF LENTH            use ERASx read/computed record length.
]WR   EQU *                Return point.
      ...
      ...                  Somewhere we would need to define ...
@WRTF DEF WRITF            Address of FMP WRITF call.
@EWRT DEF EWRIT            Address of FMP EWRIT call.
```

Of course, for the sake of efficiency, we might well set up the links to the FMP routines prior to making the pseudo-entries. The links @READ and @WRIT would be local address variables, not additional .ENTR parameter links, and would be set up this way:

```
ERASE MPX             Same as the previous ERASE sequence
      ...
      STA @MASK       until here
      LDA @REDF       This entry point uses READF calls.
      STA @READ       Set up the common link.
      LDA @WRTF       and it uses WRITF.
      STA @WRIT       set up the link.
      LDA ERASE       Same as in ERASE shown previously.
      ...             Finish off the same way as before
ERASx MPX             Same as the previous ERASF SEQUENCE
      ...
      STA @MASK       until here
      LDA @ERED       This entry point uses EREAD calls.
      STA @READ       Set up the common link.
      LDA @EWRT       and it uses EWRIT.
      STA @WRIT       set up the link.
      LDA ERASF       Same as in ERASF shown previously.
      ...             Finish off the same way as before.
```

Then instead of having to check file? prior to each READF/EREAD or WRITF/EWRIT call, we merely:

```
      ...
      JSB @WRIT,I     Call the appropriate routine.
      DEF ]WR         As before.
      ...             etc.
```

To summarize the multiple entry point rules:

1. Code the subroutine with three or more entry points, two or more of which are the real ENT names by which the routine is called, and one of which is a strictly local name which is used by .ENTR to set up the links (which may be defaulted) and through which all returns are made.

2. In the code following each real entry point, to the extent possible, set up variables, pointers, links and default links so that the subroutine's main code needs only to do its job and does not have to frequently check to see which entry point it was actually entered through.

3. Make a pseudo-entry through the pseudo-entry point, which calls .ENTR.

4. Perform the application code.

5. Reset link defaults (if necessary) and return tuo the external routine through the pseudo-entry point.

MULTI-TERMINAL ACCESS TO A REAL-TIME DATA ACQUISITION SYSTEM

Bradley Ward/Consultant Atlanta, Ga.

The RTE operating system provides very powerful programming capabilities, many of which go unnoticed, unused, and therefore unappreciated by many HP-1000 users. This article presents an application of several such capabilities, including one or two of the lesser known "hooks" and features of RTE, specifically subsystem global area (SSGA), and program scheduling on interrupt.

Before discussing the application of these RTE features, let us first examine the requirements of the software modules to be developed. The modules are intented to support program-to-program communications between a central program, and a single copy or multiple copies of a display type program, running at different terminals. The central program could be any type of process control or data acquisition program, typically running at a real-time priority level with no direct communication with any user terminal. The specific case for which the author developed the design presented in this article, was an Energy Management and Control System (EMCS), using dual HP-1000's, and interfacing to microprocessor based Field Interface Devices (FID's). The FID's provide the interface to the real world environment being monitored and controlled. The display programs encompass normal alphanumeric terminals, as well as color graphics terminals displaying status changes in real time. Due to the dedicated use of the HP-1000 in such a real time environment, a means of displaying the data on multiple terminals that minimized the loading of the processor was required. In addition to the general efficiency requirement, the following specific design criteria were to be implemented:

- continuous updating of status information on all active display terminals, as such information becomes available
- ability to input commands at any time during the execution of the display program
- ability to communicate with from zero to n copies of the display program, each running at a different logical unit
- the display program may activate or deactivate communications with the central program at any arbitrary time before, during, or after the execution of the central program

Class I/O is perhaps one of the most powerful features supported in RTE, especially for program-to-program communication applications such as the one being discussed. The beauty of class I/O is that the programs receiving the data are only active when data is available (assuming the 'NO WAIT' option is not used), and otherwise remain suspended, thereby causing no overhead due to polling or other tasks requiring processor time. The class number is the "key" to the communication between the central program and the display program. When the central detects some change requiring modification of the data being displayed by a display program, he simply puts the data onto the class queue identified by the class number shared between the central and the display program by using a class write/read call (EXEC call 20). The display program, which is suspended on a class get call (EXEC call 21), "wakes up" and receives the data from the class queue, and displays the status change.

To simplify the use of the class communications, it is recommended that each terminal display program have a unique class number assigned to it. Then, when the central desires to update status, he simply writes the data to each program's class number. In the application where the central program schedules each display program, it is a simple matter for the central to request a new class number for each program to be scheduled, passing it to the display program as one of the optional parameters on the program scheduling EXEC call. Design considerations three and four given above complicate matters, as the central cannot be sure which, or how many, of the display programs are active. Should the central continue writing to a class queue that has no one removing the data from the queue, he would soon fill up all the system available memory (SAM), causing disastrous effects on the entire operating system. To avoid such problems, the approach used is to form a table of active terminals which contains the logical unit number and the associated class number of each terminal that has a display program active. Then, when the central desires to update the system status, he need only scan the active terminal table, sending the new data to each active display program's class number.

Since the central has no means of knowing when the display program becomes active, it is up to the display program to inform him by requesting a class number through a call to EXEC, and placing an entry in the active terminal table (ATT). This requires that the ATT be located in an area of memory mutually accessible by both the central and the display programs. System common provides the easiest solution. However, due to the dedicated nature of the software system for which this was developed, system common was ruled out. Instead, subsystem global area (SSGA) was used, because it provides added protection against possible contamination of the ATT by other programs using system common. Note that the use of SSGA also allows the central to use real time system common while the display programs use background system common, or even local common.

The placement of the ATT into SSGA requires that the memory area for the ATT be set aside by a small assembler routine generated into the system. The subroutine would look something like this:

```
ASMB,R,L
       NAM ATT,30 ACTIVE TERMINAL TABLE IN SSGA
       ENT ATT
ATT    BSS 40
       END
```

This program reserves an area 40 words long in SSGA. Let us specify that the ATT will be referenced by a two dimensional FORTRAN array ITABL(I,J), where I varies from 1 to 20, allowing for up to 20 terminals to be active at one time. The J index is either 1 or 2. ITABL(I,1) is the logical unit number of the terminal in slot I of the table, and ITABL(I,2) is the class number associated with the terminal.

To link the ATT with a FORTRAN routine, labeled common will be used. As an example, a sample subroutine NOTF will be shown. NOTF is called giving a data buffer and length as arguments. NOTF's function is to send the data buffer to each active display program, as indicated by the ATT.

```
FTN4,L
       SUBROUTINE NOTF(IDATA,LEN)
       COMMON/ATT/ITABL(20,2)
 C
 C  SEND IDATA TO EACH ACTIVE DISPLAY PROGRAM.
 C
       DO 10 I=1,20
 C
 C DOES SLOT I CONTAIN AN ACTIVE TERMINAL ENTRY ??
 C
       IF(ITABL(I,1).EQ.0)GOTO10
 C
 C  YES.  PUT IDATA ONTO THE ASSOCIATED CLASS QUEUE.
 C
       CALL EXEC(20,0,IDATA,LEN,0,0,ITABL(I,2))
    10 CONTINUE
       RETURN
       END
```

Note that the linkage between ITABL and SSGA occurs because the common block name corresponds to the entry point name given to the assembler routine ATT. If the access to the ATT will be by assembler subroutines, an added level of security can be gained by naming the entry point $ATT. This prohibits the use of FORTRAN named common statements, because the "$" causes errors at compilation time. Since the labeled common block is linked to the array in SSGA, no BLOCK DATA subroutine is needed at load time. The OP,SS LOADR directive is required, however.

Now, let us examine the solution to the second design criteria listed above, the capability to input commands at any time during the execution of the display program. Due to the I/O software and hardware design of the HP-1000, at a given instant in time, either a read or a write may be pending on a given logical unit, but not both. Once a read is initiated, no writes will be processed until the request is satisfied, or the device times out. At first examination, the simultaneous solution of design criteria one (continuous display updates), and two (operator input at random intervals) would seem impossible. Fortunately, the program scheduling on interrupt feature, supported by any decent RTE compatible terminal driver, makes this possible.

Once a terminal has been enabled by the proper control call (:CT,lu,20B or :CN,lu,20B), striking a key when no read or write operations are queued to the device, causes the driver to schedule the program specified in the interrupt table at generation time (i.e. sc,PRG,PRMPT). This is best illustrated by a discussion of PRMPT and RPN, as supported by HP, under the MTM or Session capabilities of RTE.

At generation time, the interrupt table for each interactive terminal is specified as sc,PRG,PRMPT. This causes the ID segment address of the program PRMPT to be placed in the interrupt table for the specified select code. The first time the driver for the terminal is called (usually due to the :CT or :CN call in the WELCOM file), the driver examines the interrupt table and equipment table. When an ID segment address is found, RTE stores it in one of the temporary storage locations in the equipment table. For DVA05, this is EQT word 18, the second word of the 7 word EQT extension. The driver then alters the interrupt table to look as if sc,EQT,eqt# was given at generation time.

An unsolicited interrupt is an interrupt which is not expected by the driver, meaning no reads or writes are currently being processed. When an unsolicited interrupt is encountered, the driver checks the EQT status word of the interrupting logical unit to see if the specific EQT has been enabled. If it has, he then calls the EXEC to schedule the program PRMPT whose ID segment address is now held in EQT word 18 (DVA05). He also loads the B register with the address of EQT word 4 prior to scheduling the program PRMPT. This address provides the means of determining which logical unit interrupted, causing the program to be scheduled by the driver.

When PRMPT becomes active, he calls the function EQLU (described under RTE Library Subroutines in the RTE-IVB Programmer's Reference Manual) which returns the logical unit number of the interrupting device. PRMPT then writes this information onto a class queue, activating the response program, RPN. This program (RPN) issues the output "LU>" if under MTM, or "S=lu COMMAND ?" if under session (RTE-IVB). RPN then issues a read request, looking for any valid RTE command.

To solve the application problem being discussed, we need only to follow a similar path. First, let us examine the interrupt handling program we must write (our equivalent to PRMPT). We will make it a type 17 program, which specifies it will be memory resident and have SSGA access. It is best written in assembler, but is shown here written in FORTRAN.

```
FTN4,L
       PROGRAM OURIH(17,25),OUR INTERRUPT HANDLER
       COMMON/ATT/ITABL(20,2)
       INTEGER EQLU
 C
 C  WHEN THIS PROGRAM IS SCHEDULED BY INTERRUPT, THE DRIVER
 C  HAS LOADED THE B REGISTER WITH THE ADDRESS OF EQT WORD 4.
 C
       LU=EQLU(IDUM)
 C
 C  NOW LOOK AT THE ACTIVE TERMINAL TABLE TO GET THE PROPER
 C  CLASS NUMBER.
 C
       DO 10 I=1,20
       IF(LU.EQ.ITABL(I,1))GOTO20
    10 CONTINUE
 C
 C  THIS LU WAS NOT FOUND.  IGNORE THE INTERRUPT.
 C
```


```
      GOTO99
   20 CONTINUE
C
C  FOUND HIM.  SEND A MESSAGE TO THE DISPLAY PROGRAM
C  INDICATING THAT SOMEONE WISHES TO TALK TO HIM.
C  THE MESSAGE WILL BE INDICATED BY A 1 AS OPTIONAL
C  PARAMETER ONE, AND A ZERO LENGTH BUFFER.
C
      CALL EXEC(20,0,0,0,1,0,ITABL(I,2))
   99 CALL EXEC(6)
      END
```

When the display program receives the interrupt message, all he needs to do is send a "WHAT DO YOU WANT" prompt to his logical unit, followed by a read request to receive the operator's response.

It should be noted that until the operator answers this prompt, no output can be directed to the screen. Because the central program continues placing data onto this class number, the display program should either disable the communications with the central until the prompt has been answered, or make sure the terminal equipment table has a time out value associated with it, thus preventing extended periods of terminal inactivity due to a pending read.

The final task in applying this communications scheme is a way to alter the EQT table so that when the display program is running, our interrupt handler (OURIH) is used, and when it is not running, PRMPT is used in response to an unsolicited interrupt. This requires that we actually alter the EQT table upon activation and deactivation of the display program. The routines shown are for DVA05, which stores the ID segment address of the interrupt handler in word 2 of the EQT extension. These routines are FORTRAN callable, as follows:

```
CALL ACTIH(LU)  (Activates OURIH as the interrupt handler)
CALL DACIH(lu)  (Activates PRMPT as the interrupt handler)
```

```
ASMB,R,L
      NAM ACTIH,7 EQT ALTERATION SUBROUTINES
      ENT ACTIH,DACIH
      EXT IDGET,.ENTR
I1    NOP
ACTIH NOP
      JSB .ENTR
      DEF I1
      LDA I1,I
      STA LU
      JSB IDGET       GET ID SEGMENT ADDRESS OF OURIH
      DEF *+2
      DEF OURIH
      JSB ALTEQ       CALL ALTEQ TO ALTER THE EQT TABLE
      JMP ACTIH,I     RETURN.
```

```
I2     NOP
DACIH  NOP
       JSB .ENTR
       DEF I2
       LDA I2,I
       STA LU
       JSB IDGET       GET ID SEGMENT ADDRESS OF PRMPT
       DEF *+2
       DEF PRMPT
       JSB ALTEQ       CALL ALTEQ TO ALTER THE EQT TABLE
       JMP DACIH,I     RETURN.
ALTEQ  NOP
*
*  GET EQT # FOR THIS LU FROM THE DEVICE REFERENCE TABLE.
*
       STA IDAD        SAVE DESIRED ID SEGMENT ADDRESS
       LDA LU
       ADA =D-1        INDEX INTO THE DEVICE REFERENCE TABLE
       ADA DRT
       LDA A,I
       AND =B77        EQT # IS IN LOWER 6 BITS
       MPY =D15        COMPUTE EQT ADDRESS
       ADA =D-15
       ADA EQTB
       ADA =D12        COMPUTE ADDRESS OF EQT12, WHICH
       LDA A,I         CONTAINS THE ADDRESS OF THE EXTENSION.
       INA             SECOND WORD OF EXTENSION, REMEMBER.
       LDB IDAD
       JSB $LIBR       ENABLE PROGRAM TO WRITE IN SYSTEM AREA
       NOP
       STB A,I         PUT NEW ID SEGMENT ADDRESS IN EQT
       JSB $LIBX       DISABLE PROGRAM TO WRITE IN SYSTEM AREA
       DEF *+1
       DEF *+1
       JMP ALTEQ,I
 OURIH ASC 3,OURIH
 PRMPT ASC 3,PRMPT
 LU    NOP
 IDAD  NOP
 A     EQU 0
 DRT   EQU 1652B
 EQTB  EQU 1650B
       END
```

It should be noted that the above subroutine will not work for the system console. The additional software required to make the system console operate in a similar mode is to zero the SYSTY word on the base page in subroutine ACTIH, and restore it in DACIH. This code was omitted, to help simplify the above subroutine.

This outlines the specialized software needed to operate a program-to-program communications scheme which satisfies the four design criteria listed above. To implement such a system, the example subroutines shown may be used, but other subroutines will be required. As a summary, and as a guide to what other types of subroutines are needed, let us outline the steps involved in activating and deactivating a communications link between a display and the central.

1. The display program must first request a class number from the EXEC, and create an entry in the ATT, which is in SSGA.

2. The display program then must call the subroutine ACTIH to alter the equipment table of his terminal, so that our interrupt handler (which should be loaded at system generation time) is scheduled when an unsolicited interrupt occurs.

3. The display program then performs a class get on his class number, which causes him to suspend until data is sent to his queue. This data can have one of two origins:

   A. The 'posting' subroutine in the central program can place data on the queue.

   B. An unsolicited interrupt on this terminal schedules OURIH, which identifies the LU of the terminal, searches the ATT and sends a code to this terminal's class queue, causing the display program to output a "what do you want" type of message to the terminal.

4. When information is received from the queue that indicates it is time to terminate, the display program should call DACIH to restore his equipment table to its original form. Then, he should remove his entry from the ATT to prevent additional information from being added to his queue. Last of all, he should remove all information from his class queue, and return the class number to the EXEC.

5. In the central program, the data to be sent to all active terminals should be sent to a subroutine such as NOTF used in the illustration above.

It is hoped that the software described above has deepened the readers understanding and appreciation of RTE. Even if such an elaborate means of coordinating display programs is not warranted, it is quite possible that certain techniques described in the article may be adapted to other applications. A comparison of this approach with the COMMUNICATOR article "SCHEDULING DATA ACQUISITION PROGRAMS WITHOUT PROVIDING GENERAL SYSTEM ACCESS TO TERMINAL USERS" (by Frank Fulton, Volume III, Issue 3) may also increase the readers comprehension of the topics covered.

JOIN AN HP 1000 USER GROUP!

Here are the groups that we know of as of August 1980. (If your group is missing, send the Communicator/1000 editor all of the appropriate information, and we'll update our list.) We apologize for the incorrect spelling of some names in the past. They have been corrected in this issue.

NORTH AMERICAN HP 1000 USER GROUPS

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

NORTH AMERICAN HP 1000 USER GROUPS (CONTINUED)

Area	User Group Contact
Toronto	Nancy Swartz Grant Hallman Associates 43 Eglinton Av. East Suite 902 Toronto M4P1A2
Washington/Baltimore	Paul Tatavull 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

CALL FOR PAPERS
FIRST EUROPEAN HP 1000 USERS CONFERENCE
1981 THE NETHERLANDS

The "HP-RTE Gebruikers Club" (HP 1000 Users Group), in The Netherlands was founded on February 11, 1975. This year we will celebrate our first lustrum.

We would like to do this in the form of a special meeting for HP 1000 users, including not only those in The Netherlands, but users in other continents too. This includes, but is not limited to, end users and OEM's.

We are aiming for a meeting at which users can gather in great numbers to exchange experiences, and be informed of the latest technology in hardware and software, in an open forum. The conference, the first international HP 1000 users meeting in Europe, will be held in The Netherlands in April 1981. To make the meeting successful, we need your cooperation. To cover the two day agenda, we are making a **CALL FOR PAPERS,** covering the following subjects:

1. HP 1000 Users Groups
   - activities
   - services
   - .....
2. Hardware Technology
   - SOS
   - IC Technology
   - Instrument Part Technology (disc platter, printheads, etc.)
   - Computer Architecture
   - .....
3. Computational Applications
   - Graphics
   - Microprogramming
   - CAD/CAM
   - Word Processing
   - Programming Technology
   - .....
4. Data Communications
   - Remote Job Entry
   - Distributed Processing
   - Communication Drivers
   - X.25/HDLC Standards
   - Multiplexers
   - Interfacing Alien Devices/Systems
   - Network Architecture
   - Protocols
   - .....
5. Instrumentation
   - IEEE 488/HP-IB
   - Process Control
   - Data Acquisition
   - Supervisory Control
   - Instrument Interface
   - Test Systems
   - .....

6. Operating Systems
   - Real Time Executive(s)
   - Drivers
   - Languages
   - File Access Management
   - Spooling
   - Data Base Management
   - Performance Measurement/Benchmarking
   - .....

The above list is not intended to be all-inclusive. Papers in other subject areas are welcomed and encouraged. Papers must cover at least 20 and no more than 60 minutes lecture time. Submissions (3 copies) must be sent, before November 1, 1980, in the form of an abstract and summary to:

A. W. Kloosterman
University of Technology
Dept. Applied Physics
Lorentzweg 1
2628 CJ DELFT
The Netherlands

A suggested format for a submission is as follows:

Identifying Information
- Subject
- Title of Paper
- Author's name
- Address
- Disclaimer
- Duration of lecture

Abstract (maximum of 50 words)

Summary (maximum of 500 words)
- Statement of need
- Method(s) of solution
- Conclusion

Text and graphics must fit in a paper frame of 17 x 27 cm (A4) or 6.5 x 10.5 inch.

Materials submitted will not be returned. Authors will be notified of acceptance by January 1, 1981.

For all submissions a disclaimer must be attached, because we like to publish submissions in the agenda and a syllabus.

For planning purposes, we would like to ask you to send the attached form to us, before October 1, 1980. If you have any comments or suggestions, please feel free to contact us.

WATCH FOR MORE NEWS ABOUT THIS MEETING.

On behalf of the committee,

Albert R. Th. van Putten, Chairman
National Institute of Public Health
Postbox 1, 3720 BA Bilthoven
The Netherlands

LETTER OF INTEREST

This form must be sent (before October 1, 1980) to:

P.C.H. Opstal
University of Technology
Stevin Laboratory
Stevinweg 1
2600 GA DELFT
The Netherlands

This form is for the purpose of planning. By sending this form, you are not obligated to attend the conference.

☐ I expect to attend the conference

☐ I will attend the conference

☐ I/we expect to submit a paper(s) about the following subject(s): _____

☐ I/we will submit a paper(s) about the following subject(s) before _____ 1980:

Subject(s): _____

Name: _____

Institute/Company: _____

Address: _____

Country: _____ Phone: _____

ANOTHER NEW USERS GROUP!

Gary Nelson/HP Greensboro N.C.

Users of HP 1000 and HP 3000 computers in South Carolina, North Carolina, and Georgia may be interested in a new Users Group currently meeting in the Greensville, North Carolina area. Although the group welcomes anyone with general computer interests, many members are HP 1000 users.

Anyone that is interested should contact:

Henry Lucius
American Hoerchst
Greer, South Carolina
(803) 877-8741

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. 8/80

Part No. 5951-6111