DRIVER MANUAL

DRIVER MANUAL

Published by Hewlett-Packard, Palo Alto Division
November 1969

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

CONTENTS

Section		Page
I.	INTRODUCTION	1-1
II.	TYPES OF DRIVERS	2-1
III.	INPUT/OUTPUT OPERATION	3-1
	3.1 I/O Interface Structure and Operation	3-1
	3.2 The Interrupt System	3-5
IV.	NON-INTERRUPT DRIVERS	4-1
	4.1 Transferring Parameters	4-1
	4.2 Back to the GPR Driver	4-3
	4.3 Sample Voltmeter Driver	4-5
V.	NON-BCS INTERRUPT DRIVERS	5-1
	5.1 Initiator Section	5-1
	5.2 Continuator Section	5-6
	5.3 Review of Interrupt Drivers	5-9
VI.	ABSOLUTE DRIVERS FOR BASIC	6-1
	6.1 Paging	6-1
	6.2 Locating Your Driver	6-3
	6.3 Linkage Table	6-5
	6.4 Body of the Driver	6-6
	6.5 Ending Service Table	6-7
	6.6 Review of Absolute Drivers for Basic	6-8
	6.7 BASIC Drivers	6-9
	6.8 Protecting the BASIC Compiler	6-17

Appendices

A	Relocatable GPR Driver Modified to Take Multiple Readings	A-1
B	.ENTR Routine for BASIC Drivers	A-2

ILLUSTRATIONS AND TABLES

Figure		Page
3-1	Simplified Interrupt Diagram	3-0
3-2	Photoreader Operating Elements, Simplified	3-2
3-3	Tape Punch Operating Elements, Simplified	3-3
3-4	Tape Duplicator Program.	3-4
3-5	Interrupt Priority Assignment by Interface Location	3-5
3-6	Priority System, Simplified	3-6
3-7	Driver Operating Sequence	3-7
3-8	Strain Gage Data Processing Flowchart	3-9
4-1	3440A Range and Function Input Format	4-6
5-1	Data Read-Write Program Flowchart	5-12
6-1	Division of 8K Memory into Pages	6-2
6-2	Format of First Word of Linkage Table Entry	6-5
6-3	How Pointers Delimit Storage Areas in Core	6-18

Table		
6-1	Address Reference Chart for Integrating Absolute Drivers into BASIC Compilers	6-21

I. INTRODUCTION

Talking to instrumentation through HP computers is a simple task. All that is needed is an input/output interface card, already designed for you by HP, and a software driver, which may or may not have been developed. This booklet will discuss the latter.

Understanding drivers is useful because it makes possible programming of an instrument that has never been tied to a computer before. An understanding of drivers is also essential if you want to modify existing drivers to satisfy requirements of your own special application.

There is nothing mysterious or difficult about the piece of software called a driver. It is just another part of a program. We give drivers special treatment here because they are used repetitively and they interact with the interrupt system.

Most input/output devices are much slower than the computer. For this reason, drivers utilize the interrupt system of HP computers. The interrupt system allows us to use a device without making the computer wait for that device. We can, for example, tell a voltmeter to take a reading and then have the computer process some previously acquired data. When the voltmeter actually gets the data, the computer is interrupted and forced to service the voltmeter. This process will be discussed in detail in section III.

II. TYPES OF DRIVERS

Before we get into the actual writing of drivers, we should discuss the different types. Along with this discussion we must also standardize on various terms used relating to drivers.

Basically there are two main types of drivers — relocatable and absolute. Relocatable drivers are named such because they are "positioned" in core at a location determined by the relocating loader. These drivers are FORTRAN, ALGOL, or relocatable Assembler language callable, which means that if you will be programming in these languages, you must write relocatable drivers. The second main type of driver is an absolute driver. These are "positioned" at a particular location in core. If you plan on programming instruments in BASIC, then you will need to write absolute drivers.

Relocatable drivers are of two main types — BCS and non-BCS drivers. The choice of these names may add to confusion but since they have evolved and are being used, we will try to stick by them. The main difference between the two is that the BCS driver uses IOC (Input Output Control) whereas the non-BCS driver does not use IOC. IOC, you recall, is the traffic cop that tells drivers where the interface card is located, where the driver is in core, whether or not the drivers are busy and so on. Non-BCS drivers, although they do not use IOC, can perform all of the above functions. Often the difference between the BCS and the non-BCS driver is very small.

BCS drivers must perform many extra functions automatically, such as:

1. They are device independent
2. They provide and update status check
3. They permit use of buffered IOC
4. They check to see if the call was valid
5. They provide a system clear routine

Non-BCS drivers are much more liberal. They require that you do only what your heart and driver require, so you can get into trouble applying a non-BCS driver to an application for which it was not intended. In those applications for which they *are* intended, most people find that the non-BCS drivers are easier to formulate and integrate into their system because those drivers do no more than is needed to get the job done. The BCS driver may, by definition, do more than the application requires.

This booklet does not discuss BCS drivers. That type of driver is described in *"A Pocket Guide to Interfacing HP Computers."* For those adventurous souls, however, who love to "wing it," proceed on to learn the diabolical art of non-BCS drivers.

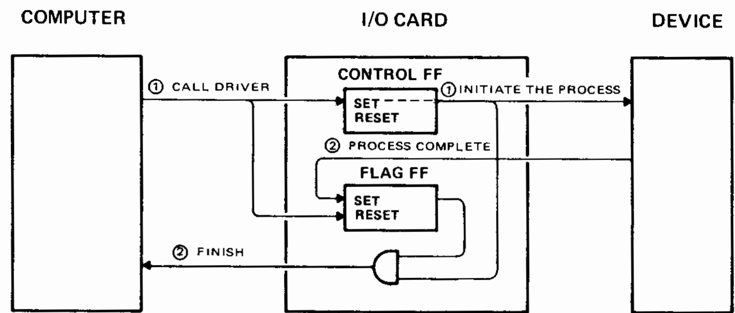


Figure 3-1. Simplified Interrupt Diagram

III. INPUT/OUTPUT OPERATION

In order to write any type of driver, it is necessary to have a working knowledge of the input/output section of the computer. All HP computers and interface cards operate alike and have the same general I/O and interrupt system structure.

3-1 I/O INTERFACE STRUCTURE AND OPERATION

Most HP I/O interface cards contain two flip-flops that control their operation (Figure 3-1, facing).

The first of these is the control flip-flop. This can be thought of as a big switch that starts the peripheral device (voltmeter, scanner, A/D converter) doing its job. The other is the flag flip-flop which signals the completion of the process (data ready, voltmeter encoded, conversion completed). It can be visualized by its name — the device is waving a flag saying, "I've done my job, service my interrupt."

The sequence of events and their corresponding assembly language instructions are simple: first, we must set the control flip-flop and then clear the flag flip-flop as in Step 1 of Figure 3-1. This initiates the process we are trying to perform. We clear the flag flip-flop in the same statement that we set the control flip-flop, so that we will have a means of knowing when the process is completed. If the flag was set and we failed to clear it, the computer would be tricked into believing that the process was already completed. The assembly language coding to set the control and clear the flag is:

```
STC   IO,C
```

Where IO is the octal I/O slot number of the selected device. Upon completion of the process, the flag flip-flop gets set by the peripheral device, as shown in Step 2 of Figure 3-1. When the flag is set, either of two things can happen:

1. If the interrupt system is disabled, we can test the flag with a skip-if-flag-set command (SFS)

2. If the interrupt system is not disabled, the program will be interrupted automatically.

These two events will be discussed in detail in the next sections.

3.1.1 Photoreader Example

In order to visualize what actually happens in a driver, let's follow through the process using the photoreader and punch as an example. See the simplified diagram of the photoreader, below.

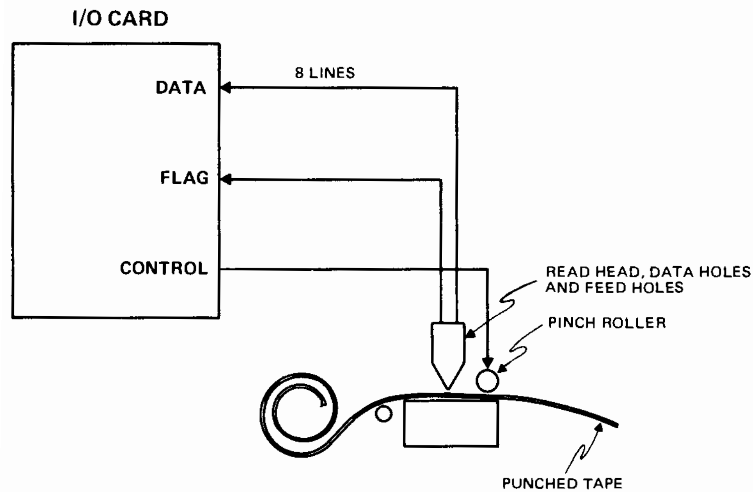


Figure 3-2. Photoreader Operating Elements, Simplified

The operation of the photoreader is simple, involving the elements shown in Figure 3-2.

1. It feeds the tape by closing the pinch roller.
2. It looks for the small feed hole.
3. It inputs the data.

The set control and clear flag instruction (STC IO,C) closes the pinch roller. The sensing of the feed hold sets the flag and opens the pinch roller. A load into the A-register (LIA I/O) instruction inputs the data.

The coding to drive the photoreader located physically in I/O slot 10_8 is as follows:

STC	**10B,C**	Sets the control flip-flop and clears the flag flip-flop on I/O card 10_8, causing the pinch roller to close.
SFS	**10B**	Tests the flag flip-flop in I/O slot 10_8; if it is set, it skips the next instruction; if not, executes the next instruction. In this case the flag is not set until the feed hole is sensed.
JMP	***-1**	Causes an unconditional jump to the previous instruction. In this example, we are looping on the skip flag set instruction until the feed hole is sensed.
LIA	**10B**	Loads data from I/O slot 10_8 into the A-register. In this example, we bring in the data from the photoreader.

3.1.2 Punch Example

That concludes the driver for the photoreader, but in order to show you something useful, let's also discuss a punch driver. (See Figure 3-3.)

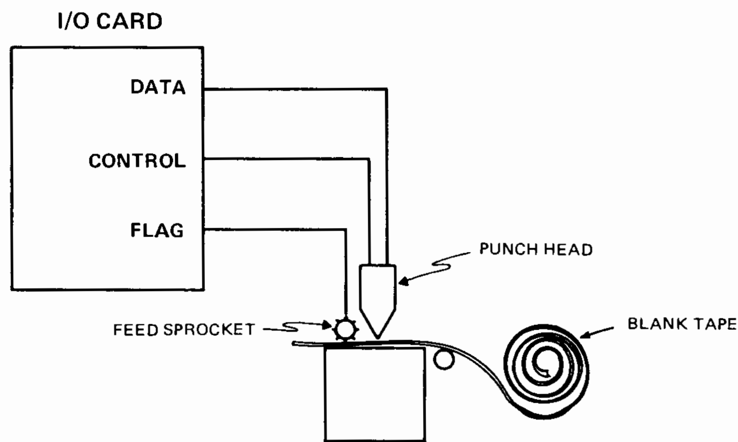


Figure 3-3. Tape Punch Operating Elements, Simplified

The sequence of operation is similar to that of the photoreader.

1. Set the punch solenoids.
2. Punch the tape.
3. Move the tape.

In Assembler language the driver for the punch located in slot 11_8 is as follows:

OTA 11B Outputs the contents of the A-register to the I/O card in slot 11_8. In this example, it sets up the punch solenoids.

STC 11B,C Sets the control and clears the flag on the I/O card in slot 11_8. Here this command causes the tape to be punched.

SFS 11B Tests the flag in I/O slot 11_8. In this case we are checking to see if the punching is completed and the tape has been moved.

JMP *-1 An unconditional jump to the previous instruction. In this example we are looping until the punch has finished punching.

We now have demonstrated two drivers that can actually be used. Together they form a duplicator program. It is beneficial at this time to actually go to your computer and enter them through the switch register. The program is listed in Figure 3-4.

LABEL	OP CODE	OPERAND	REMARKS	LOCATION	CONTENTS
LOOP	STC	10B,C		1000	103710
READ	SFS	10B		1001	102310
	JMP	READ		1002	025001
	LIA	10B	BRING IN DATA	1003	102510
	OTA	11B	OUTPUT TO PUNCH	1004	102611
	STC	11B,C		1005	103711
PUNCH	SFS	11B		1006	102311
	JMP	PUNCH		1007	025006
	JMP	LOOP		1010	025000

Figure 3-4. Tape Duplicator Program

For those of you who are not accustomed to entering programs through the switch register, the procedure is: load address 1000_8, then load memory with the contents listed above; place a NOP (all zero's) in memory addresses 10_8 and 11_8; and finally load address 1000_8 again. Then push Preset and Run with tape in the photoreader. The tape should be read and a duplicate of it punched. I'm sure you're now wondering why we placed NOP's in memory locations 10_8 and 11_8. This is a tricky way to introduce the discussion of the priority interrupt system.

3.2 THE INTERRUPT SYSTEM

HP computers feature a hardware priority interrupt system. This enables us to communicate with external devices, which are typically much slower than the computer, without making the computer wait. This means that we can be processing data at the same time we are acquiring it. A typical application of processing under interrupt is a digital voltmeter used to acquire data at some rate, say 40 readings/second, while in between readings we calculate the RMS value, compare to high-low limits, or similar calculations. Whenever a reading comes in, the computer is interrupted, the data brought in, another measurement initiated, and control is returned to the main program at the place it was interrupted; processing of the data then continues until another interrupt occurs.

Priorities of the interrupts are assigned by the location of the interface cards in the computer's I/O section, shown below in Figure 3-5.

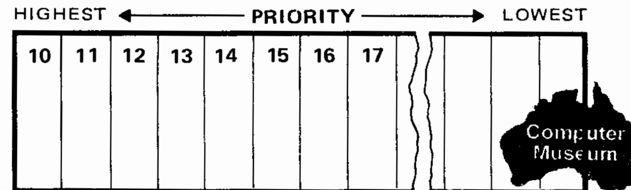


Figure 3-5. Interrupt Priority Assignment by Interface Location

Priorities are assigned from left to right, left having the highest priority, or in terms of the I/O slot numbers, the lower the slot number, the higher the priority. Very simply stated, the priority system is a line running into and out of all the I/O cards as in Figure 3-6.

If, for example, the device in slot 11 interrupts, the priority line (bold) coming out of that card is disabled. This prohibits any lower priority device from interrupting. When the interrupt system is used, four conditions must be true simultaneously in order for this interrupt to disable the priority line:

1. The interrupt system must be enabled.
2. The flag flip-flop must be set.
3. The control flip-flop must be set.
4. No higher-priority device has disabled the priority line.

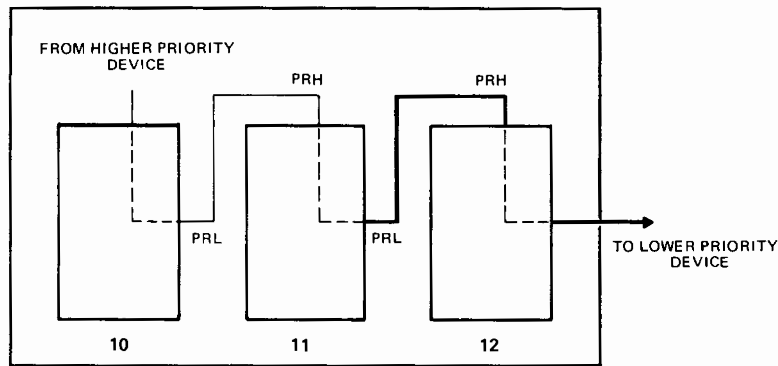


Figure 3-6. Priority System, Simplified

The priority line remains disabled until one of these four conditions is removed. This is the reason we clear the control flip-flop or the flag flip-flop before exiting a driver. The flag flip-flop is usually cleared by a set control, clear flag instruction when multiple measurements are desired. The control flip-flop is cleared by a clear control command before exiting the driver the final time.

The operation of the priority interrupt system is straightforward. Six conditions must be met to cause an interrupt:

1. The interrupt system is enabled.
2. The flag and flag buffer flip-flops of the specified device interface card are set.
3. The control flip-flop of the specified device interface card is set.
4. No priority-affecting instruction (STF, CLF, STC, and CLC) is in progress.
5. No higher priority devices satisfy conditions 1 through 4.
6. No JSB indirect, JMP indirect, EAU*, or DMA Instruction is being executed.

From a user's standpoint, the main point is that *both the control and flag flip-flops must be set* to cause an interrupt. The control is set by the programmer to initiate the process; the flag is set on the I/O card by the device itself to signify the completion of the process. Together they cause an interrupt.

**Extended arithmetic unit (optional).*

An interrupt forces execution of one instruction at a specified location in memory, which is called a "trap cell." The address of the trap cell is an address number that corresponds to the I/O slot number, i.e., slot number 15_8 is associated with memory location 15_8 (trap cell 15_8).

The trap cell can contain any of 72 instructions; but, for interrupt drivers, it will usually contain a jump to a subroutine indirect (JSB SUB,I), which will perform the necessary function of inputting or outputting data. Upon completion of this routine, the computer is returned to pre-interrupt status, and processing is continued.

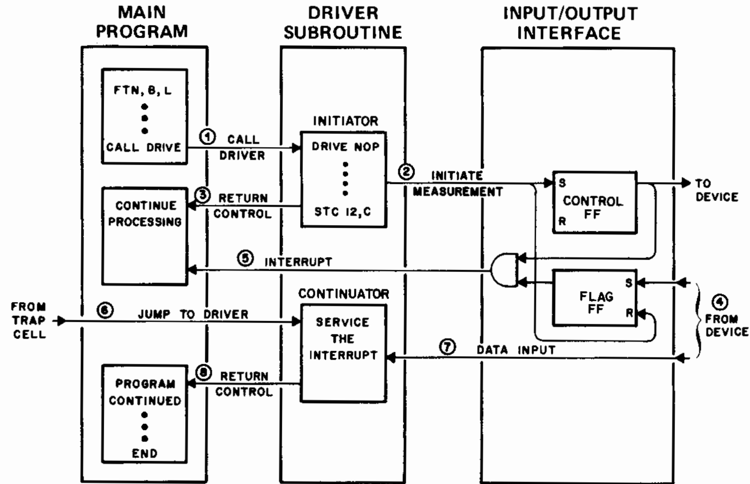


Figure 3-7. Driver Operating Sequence

A general review of what happens when using the interrupt drivers is shown visually in Figure 3-7 and summarized below.

1. The main program calls the driver subroutine.
2. The measurement is initiated by setting the control flip-flop and clearing the flag flip-flop on the interface card serving the selected peripheral device.
3. Control is returned to the main program and processing is continued until an interrupt is received.

4. When the selected device has performed its function, the flag is set.
5. The main program is then interrupted, and the trap-cell instruction is executed.
6. The trap-cell instruction causes the computer to jump to a subroutine which services the device that interrupted.
7. Control is returned to the main program at the point and with the same status as before the interrupt occurred.
8. Processing is continued.

A practical example of processing under interrupt is a strain gage rosette analysis. For this application, we need three pieces of X, Y, and Z axis strain gage data, before we can convert voltage readings into useful engineering units such as ft-lbs. of force. Let us imagine that we have 20 sets of 3 strain gages on an airplane wing that is being tested in a wind tunnel. Here, processing under interrupt would be very valuable in that we could convert some data while waiting for more data to be input. The sequence of events would proceed as diagrammed in Figure 3-8.

We initiate the measurement by calling the driver. We then wait until we have enough data, in this case 3 readings, to do some processing. When we have 3 readings, we can convert this data to useful information. While the program is converting this data, it is being interrupted to input more data. By the time we have completed the conversion, we may have input another set of three data points. In that case, we continue processing and never notice the relative slowness of the input device. There are many applications where processing and inputting under interrupt is valuable. Generally, these are applications where considerable processing or data reduction is needed in a system with relatively slow I/O devices.

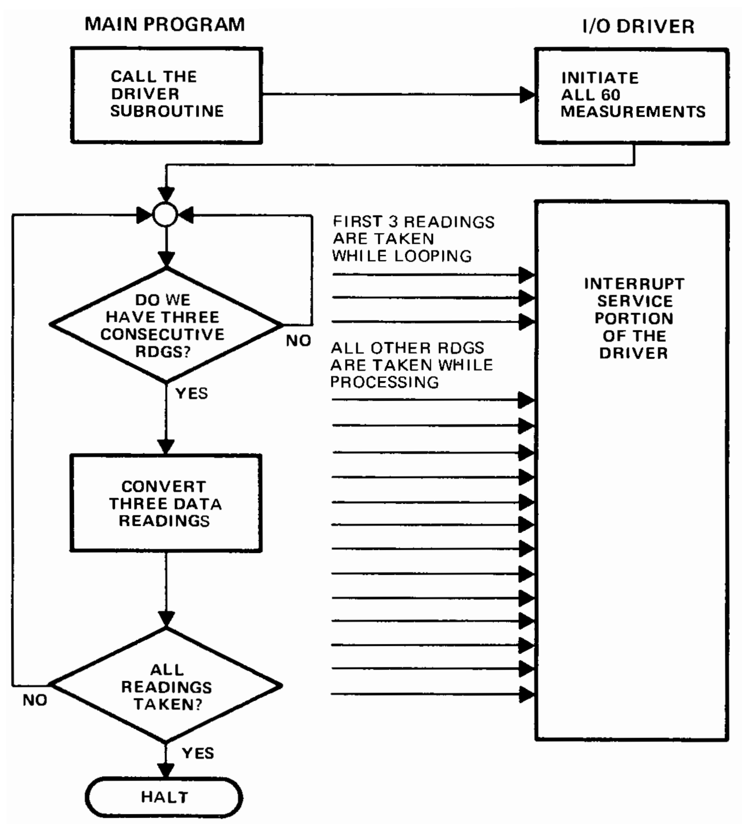


Figure 3-8. Strain Gage Data Processing Flowchart

IV. NON-INTERRUPT DRIVERS

Non-interrupt drivers are straightforward. We have already written and demonstrated two of them in the previous section. Here we will discuss how you link these drivers to the calling program using an example of the Duplex (general purpose) register.

Before we plunge into the discussion of the actual driver, let's review a few Assembler language instructions in case you haven't been doing your daily Assembler language exercises.

First there is the Assembler language control statement. This is briefly:

```
ASMB, B, L, R
```

B — Binary output
L — List output
R — Relocatable program

If you did not understand the above statement, stop reading this booklet and yet yourself the HP Assembler Manual.

Secondly, we must name our program with the NAM instruction:

```
NAM  GPR
```

In this case we call it "GPR" for general purpose register.

Next we must define our entry points and our external locations with the ENT and the EXT instructions:

```
EXT  .ENTR
ENT  GRP
```

In this case GPR is defined as an entry point and .ENTR is defined as an external location to the driver. Defining .ENTR as external concludes our brief review of Assembler language and makes a smooth transition into the discussion of transferring parameters.

4.1 TRANSFERRING PARAMETERS

Transferring parameters can be as difficult or as easy as you wish to make it. Since I'm of the easy school, we'll use a routine that you already have on your FORTRAN/ALGOL library tape. This routine is sneakily called ".ENTR." In case you are interested, in your *Pocket Guide to Hewlett-Packard Computers,* there's a section labeled "Program Library Reference Manual" that contains the description of a group of useful routines, including .ENTR, that can really simplify your programming. Additional information on use of .ENTR is also in Appendix C of the section of the pocket guide which is labelled "FORTRAN Reference Manual".

The use of .ENTR is relatively mechanical. First, you list the parameters in the label field with corresponding NOP's in the op code field, in the order in which they were sent to the driver. After the list of parameters, place the entry point of the driver, again with a NOP in the op code field. Now place a jump subroutine instruction to .ENTR followed by a DEF instruction which refers back to the first parameter listed. If this all sounds confusing, keep reading. If we want to be able to transfer three parameters to the driver with the following call from FORTRAN:

```
CALL GPR (DATA, NUMB, IEND)
```

we would "receive" these in the driver with the following series of instructions:

```
DATA   NOP )
NUMB   NOP  } ———————Parameters listed with NOP's
IEND   NOP )
GPR    NOP             Entry point of driver
       JSB   .ENTR     Jump to .ENTR
       DEF   DATA      Pointer to first parameter
```

Now, it's your turn. Write the routine to transfer two parameters to the driver named DRIVE. The FORTRAN call statement is:

```
CALL     DRIVE (VALUE, ICHAN)
```

Your coding should be:

```
VALUE  NOP }
ICHAN  NOP }——————— Parameters to be transferred
DRIVE  NOP             Entry point of driver
       JSB   .ENTR     Jump to .ENTR
       DEF   VALUE     Pointer to the first parameter
```

Now that we have the mechanics out of the way, it's time to understand one major point about .ENTR: *.ENTR transfers the addresses of the parameters, not the values of the parameters.* If you desire the contents of an integer parameter, such as ICHAN, you can obtain it by merely loading the A-Register indirect with the following instruction:

```
       LDA   ICHAN,I
```

To get the contents of a real (two-word) parameter, such as VALUE, you should use the DLD (Double Load) pseudo instruction, as follows:

```
       DLD   VALUE,I
```

The use of DLD, and the companion DST.(Double Store) instruction is discussed in the "Assembler Reference Manual" section of the *Pocket Guide to Hewlett-Packard Computers*, under the heading 'Arithmetic Subroutine Calls'.

Now, let's review the use of .ENTR:

1. List the parameters to be transferred in the right order.
2. List the entry point of the driver.
3. Jump subroutine to .ENTR
4. Point to the first parameter with the DEF instruction.

4.2 BACK TO THE GPR DRIVER

So far, we have discussed the coding necessary to establish entry and external points of a driver and how to transfer parameters. If we assume a FORTRAN call of:

```
CALL    GPR (IDATA)
```

the coding thus far is:

```
ASMB,B,L,R                  Control statement
        EXT   .ENTR         External locations
        ENT   GPR           Entry point
DATA    NOP                 Parameter
GPR     NOP                 Entry point
        JSB   .ENTR         Jump to .ENTR
        DEF   DATA          Pointer to first parameter
```

Now we can proceed into the meat of the driver. A duplex or general purpose register, you recall, can input or output 16 bits of digital data. In this example, we are going to input 16 bits and store this information in a location called IDATA.

The first order of business is to place a NOP (later we will use a clear control command) in the selected device's trap cell so that nothing will happen when the interrupt occurs. To do this, we write:

```
        CLA                 Clears the A-Register
        STA   IO            Stores the A-Register in memory location IO
```

You will notice that from here on we will use the label IO to refer to an octal I/O slot number, instead of the actual number as we did in the photoreader-punch example. This convention makes it a simple matter to modify the driver for another I/O slot number since we define the slot number with an EQU instruction in the program as shown:

```
IO      EQU   12B
```

The above instruction defines IO to be 12_8 wherever it's used in the program:

Now that we have made certain that nothing would happen as a result of interrupt, we are ready to initiate the process. To do this, you recall, we use a set control clear flag instruction:

```
STC   IO,C
```

Next we want to test the flag to see if the data is ready to be transferred. We do this with a skip-if-flag-set instruction:

```
SFS   IO
```

With this coding, we skip the next sequential program instruction if the flag is set. The next instruction will be executed if the flag is not set. Since we want to wait for data before proceeding, we use a jump back to the skip-if-flag-set instruction:

```
JMP   *-1
```

We exit the above loop when the data is ready. We are now ready to input the data with a load into the A-Register instruction from the appropriate I/O slot:

```
LIA   IO
```

Next, we are required to store this data in the memory location contained in the parameter DATA. We have already transferred the address of DATA, using the .ENTR routine. To store the A-Register in address contained at location DATA, we use the following instruction:

```
STA   DATA,I
```

We have now completed the major tasks of the GPR driver. Before exiting our driver, we must remember to clear the control flip-flop, which will re-establish the priority string. To do this, we merely clear the control on the selected I/O slot:

```
CLC   IO
```

To return to the calling program with the reading neatly tucked away in the location DATA, we jump indirect through the entry point of the driver. The entry point just happens to contain the appropriate return point of the main program. For this fact, we again need to thank the .ENTR routine. So to return, we write:

```
JMP   GPR,I
```

The driver, as a whole, as follows:

```
ASMB,B,L,R                      Control statement
        NAM  DRIVE              Program name
        EXT  .ENTR              External location
        ENT  DRIVE              Entry point
IO      EQU  12B                I/O slot definition
DATA    NOP        )
DRIVE   NOP        }            Parameter transfer
        JSB  .ENTR }
        DEF  DATA  )
        CLA        }            Disables the interrupt by NOPing the trap cell
        STA  IO    }
        STC  IO,C               Initiates device
        SFS  IO    }            Waits for reading
        JMP  *-1   }
        LDA  IO                 Input data
        STA  DATA,I             Stores data in calling program
        CLC  IO                 Re-establishes priority string
        JMP  DRIVE,I            Returns to calling program
        END
```

A simple example of a FORTRAN program that would use the above driver is one that calls the driver and prints the results on the teleprinter. This program follows:

```
FTN,B,L,A
        PROGRAM EXAM
        CALL DRIVE (IDATA)
        WRITE (2, 100) IDATA
100     FORMAT ("DATA" I6)
        END
```

The above program would interpret the bit formation input on the GPR card as an integer number and would output it as such.

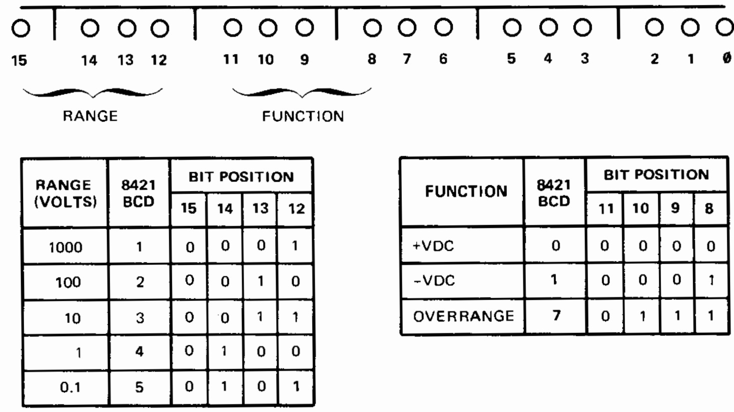
4.3 SAMPLE VOLTMETER DRIVER

Let's conculde this section on non-interrupt drivers with a relatively complex example (pages 4-7 through 4-10) of a FORTRAN callable driver for an HP 3440 voltmeter *that is equipped for dc voltage measurements only*. This example is more difficult because it has a lot more goodies added to it than does a simple non-interrupt driver. This driver has the following additional capability:

1. It aborts the measurement if no reading is returned within 10 seconds. When this occurs, a "-1" is placed in the A-Register and the DATA is set equal to Ø.Ø.

2. If a normal reading is taken, the data is converted from its 8-4-2-1 BCD code to its floating point binary equivalent.
3. If an overload condition occurs, a "+1" is placed in the A-Register and data is returned as Ø.Ø.
4. 32-bits are input. The first input instruction (LIB in this example) brings in the four data digits; the second input instruction (LIA in this example) brings in the function and range as shown in Figure 4-1.

At this point, your understanding of driver-writing as so far presented can be nailed down by working on your own non-interrupt driver. If you have no specific application in mind, a good exercise is to write a non-interrupt driver for the photo reader and a FORTRAN program that calls it and prints out the data on the teleprinter. Further experience can be gained by adding a method of timing the process and aborting operation with all zeros returned if no tape is in the photo reader. With this kind of experience under your belt you will be ready to proceed with the next section, which deals with interrupt drivers.



RANGE (VOLTS)	8421 BCD	BIT POSITION			
		15	14	13	12
1000	1	0	0	0	1
100	2	0	0	1	0
10	3	0	0	1	1
1	4	0	1	0	0
0.1	5	0	1	0	1

FUNCTION	8421 BCD	BIT POSITION			
		11	10	9	8
+VDC	0	0	0	0	0
-VDC	1	0	0	0	1
OVERRANGE	7	0	1	1	1

Figure 4-1. 3440A Range and Function Input Format

```
0001                 ASMB,B,L,R
0001  00000                  NAM M3440
0002                         ENT MEAS
0003                         EXT .ENTR,FLOAT,.FMP,..FCM
0004*
0005*     CALLED FROM FORTRAN AS
0006*
0007*        CALL MEAS (DATA)
0008*
0009*        WHERE DATA = A FLOATING POINT VARIABLE NAME WHICH WILL BE
0010*      SET TO THE SIGNED VALUE OF DATA FROM THE 3440 DVM
0011*
0012*        DATA WILL BE SET EQUAL TO ZERO IF THERE IS AN OVERLOAD OR
0013*        IF NO MEASUREMENT IS RETURNED WITHIN TEN SECONDS.
0014*
0015*     MAY BE CALLED FROM FORTRAN AS
0016*
0017*        IF (MEAS(DATA)) 10,20,30
0018*
0019*           TRANSFER TO BRANCH 10 = DVM FAILED TO RESPOND
0020*           TRANSFER TO BRANCH 20 = NORMAL MEASUREMENT
0021*           TRANSFER TO BRANCH 30 = OVERLOAD
0022*
0023*
0024*     PARAMETER PICK UP AND ENTRY
0025*
0026  00013          .3440 EQU 13B        I/O CHANNEL LOCATION OF DSI CAR
0027  00000 000000  DATA  BSS 1
0028  00001 000000  MEAS  NOP             ENTRY POINT
0029  00002 016001X       JSB .ENTR       PARAMETER PICK-UP ROUTINE
0030  00003 000000R       DEF DATA
0031*
0032*      SUPPRESS INTERRUPT BY STORING A "NOP" IN THE TRAP CELL
0033*
0034  00004 002400        CLA
0035  00005 006400        CLB
0036  00006 070013        STA .3440
0037*
0038*     INITIATE MEASUREMENT
0039*
0040  00007 103713        STC .3440,C
```

```
0042*
0043*       WAIT UP TO TEN SECONDS FOR THE DATA
0044*
0045*      IF DATA IS RETURNED GO TO "INPUT"
0046*      IF NO DATA IS RETURNED GO TO "FAIL"
0047*      INCREMENT A REGISTER EVERY 100 MILLISECONDS FOR DISPLAY
0048*
0049  00010 006004  WAIT  INB             10 MICROSECOND WAIT LOOP
0050  00011 054004B       CPB .10K        10000 LOOPS = 100 MILLISECONDS
0051  00012 026016R       JMP *+4        INCREMENT A-REGISTER
0052  00013 102313        SFS .3440       DATA YET?
0053  00014 026010R       JMP WAIT        NO,WAIT
0054  00015 026032R       JMP INPUT       YES, ACCEPT DATA
0055*
0056  00016 002004        INA             A REGISTER DISPLAY
0057  00017 050003B       CPA D100        10 SECONDS
0058  00020 026023R       JMP FAIL        GIVE UP AFTER 10 SECONDS
0059  00021 006400        CLB             RESET B-REGISTER
0060  00022 026010R       JMP WAIT
0061*
0062*      FAIL
0063*
0064  00023 002400  FAIL  CLA
0065  00024 006400        CLB
0066  00025 104400        DST DATA,I      SET DATA TO ZERO
      00026 100000R
0067  00027 003400        CCA             LEAVE -1 IN THE A-REG
0068  00030 106713        CLC .3440       KEEP PRIORITY OPEN
0069  00031 126001R       JMP MEAS,I
0070*
0071*      INPUT
0072*
0073  00032 106713  INPUT CLC .3440
0074  00033 106513        LIB .3440       4-DATA DIGITS
0075  00034 102513        LIA .3440       RANGE AND FUNCTION
0076  00035 001727        ALF,ALF
0077  00036 070006B       STA RANDF
0078  00037 074005B       STB IDATA
```

```
0080*
0081*      CONVERT 8-4-2-1 DATA TO FLOATING POINT
0082*
0083*     IF BIT 0 OF RANDF = 1 THEN VOLTAGE IS NEGATIVE
0084*
0085  00040 006400        CLB
0086  00041 002011        SLA,RSS
0087  00042 064012B       LDB RSS1
0088  00043 076056R       STB SKIP
0089*
0090*     BITS 4-5 OF RANDF = NEGATIVE POWER OF 10 (1,2,3)
0091*
0092  00044 010010B       AND M60        A = 60,40,20,0
0093  00045 001323        RAR,RAR
0094  00046 001300        RAR
0095  00047 040007B       ADA DRANG      A POINTS TO .001,.01,.1,0
0096  00050 072055R       STA FACTR      MULTIPLY DATA BY FACTR
0097*
0098*     DATA CONVERSION
0099*
0100  00051 060005B       LDA IDATA
0101  00052 016106R       JSB BCDTB      8-4-2-1 TO BINARY
0102  00053 016002X       JSB FLOAT      TO FLOATING POINT
0103  00054 016003X       JSB .FMP
0104  00055 000076R FACTR DEF RANGE      MULTIPLY DATA BY FACTR
0105  00056 002001  SKIP  RSS            SKIP IF POSITIVE
0106  00057 016004X       JSB ..FCM
0107  00060 104400        DST DATA,I
      00061 100000R
0108*
0109*     SET A-REG TO 0 FOR NORMAL MEASUREMENT, 1 FOR OVERLOAD
0110*
0111  00062 060006B       LDA RANDF
0112  00063 010011B       AND M17
0113  00064 050011B       CPA M17
0114  00065 026070R       JMP OVLOD
0115  00066 002400        CLA
0116  00067 126001R       JMP MEAS,I   NORMAL EXIT
0117*
0118  00070 002400  OVLOD CLA
0119  00071 006400        CLB
0120  00072 104400        DST DATA,I
      00073 100000R
0121  00074 002004        INA
0122  00075 126001R       JMP MEAS,I   OVERLOAD
0123*
0124  00076 040000  RANGE DEC 1.,.1,.01,.001
      00077 000002
      00100 063146
      00101 063373
      00102 050753
      00103 102765
      00104 040611
      00105 033757
```

```
0126*
0127*     BCDTB
0128*
0129  00106 000000  BCDTB NOP            ENTER WITH 8-4-2-1 BCD IN A-REG
0130  00107 064013B       LDB DM16     BIT COUNTER
0131  00110 074002B       STB CTR     LOOP COUNTER
0132  00111 064000B       LDB DTABL   POINTER TO DECIMAL LIST
0133  00112 074001B       STB PTR        POINTER TO DECIMAL LIST
0134  00113 006400        CLB            FORM BINARY IN B-REGISTER
0135*
0136  00114 002020  LOOP  SSA            IS A BIT ON
0137  00115 144001B       ADB PTR,I      YES, ADD APPROPRIATE VALUE
0138  00116 001200        RAL            NEXT BIT
0139  00117 034001B       ISZ PTR        NEXT VALUE
0140  00120 034002B       ISZ CTR        DONE?
0141  00121 026114R       JMP LOOP       NO
0142  00122 060001        LDA 1          ANSWER IN A-REG
0143  00123 126106R       JMP BCDTB,I    RETURN
0144*
0145  00124 017500  TABLE DEC 8000,4000,2000,1000,800,400,200,100
      00125 007640
      00126 003720
      00127 001750
      00130 001440
      00131 000620
      00132 000310
      00133 000144
0146  00134 000120        DEC 80,40,20,10,8,4,2,1
      00135 000050
      00136 000024
      00137 000012
      00140 000010
      00141 000004
      00142 000002
      00143 000001
0147*
0148*     BASE PAGE CONSTANTS
0149*
0150  00000               ORB
0151  00000 000124R DTABL DEF TABLE
0152  00001 000000  PTR   BSS 1
0153  00002 000000  CTR   BSS 1
0154  00003 000144  D100  DEC 100
0155  00004 023420  .10K  DEC 10000
0156  00005 000000  IDATA BSS 1
0157  00006 000000  RANDF BSS 1
0158  00007 000076R DRANG DEF RANGE
0159  00010 000060  M60   OCT 60
0160  00011 000007  M17   OCT 7
0161  00012 002001  RSSI  RSS
0162  00013 177760  DM16  DEC -16
0163                      END
**  NO ERRORS*
```

V. NON-BCS INTERRUPT DRIVERS

Did you ever want to be a juggler? Then you'll enjoy interrupt drivers because they give the appearance of many processes occurring at the same time. Actually only one program is being executed at any given instant, but the computer's ability to interrupt itself really livens the issue.

Drivers that operate under interrupt control are divided into two main parts. The first part initiates the measurement and returns control to the main program. This section is appropriately named the "Initiator." The second section services the interrupt, transfers the data, and restores the computer to its pre-interrupted status. This section basically continues the measurement routine and is therefore referred to as the "Continuator." These two sections will be discussed in detail.

5.1 INITIATOR SECTION

The initiator section is exactly like any other subroutine. It is called by the main program, it can use any or all of the FORTRAN/ALGOL library, and control is returned to the calling program.

The initiator section of any driver should:

1. Transfer all parameters needed such as (a) the address where the data is to be stored, (b) the number of readings to be taken, and so on.
2. Check to see if the driver is busy.
3. If not busy, set driver busy flag.
4. Transfer all the calling parameters to a buffer area.
5. Establish a "reading taken indicator."
6. Establish the interrupt linkage; i.e., place a JSB to the continuator section of the driver in the "trap cell."
7. Tell the device to start operating; i.e., set its control flip-flop and clear its flag flip-flop.
8. Return control to the calling program.

Now let's see if we can apply what we discussed to the task of writing the initiator section of a GPR (general purpose register) driver. The GPR interface card, you recall, can input and/or output 16 bits of information. In this example, let's use it to input one reading of 16 bits. Let's assume the GPR card is physically located in I/O slot 12_8.

5.1.1 Transfer Parameters

There are various ways we handle transferring parameters, but let's stick to the .ENTR routine already learned. Remember the steps:

1. List the parameters in the appropriate order in the label field with NOP's in the operand field.
2. List the entry point of the initiator section of the driver in the label field with a NOP in the operand field.
3. Jump subroutine to .ENTR followed by a DEF instruction pointing to the first parameter to be transferred.

Now, It's time to stress an important point: The jump subroutine to .ENTR *must immediately follow* the labeling statement of the driver, so we cannot check to see if the driver is busy before we pass the parameters. For our GPR driver example, we will transfer two parameters: DATA and IEND, the function of which will be explained later.

```
ASMB,B,L,R,T            Control statement
        NAM  GPR        Name of program
        EXT  .ENTR      External location
        ENT  GPR        Entry point
IO      EQU  12B        I/O slot definition
IDATA   NOP          )
IEND    NOP          |
GPR     NOP          >  Parameter transfer
        JSB  .ENTR   |
        DEF  IDATA   )
```

5.1.2 Check the Driver Busy Flag

After transferring the parameters, we must next check the driver busy flag. The busy flag is just a core location we have labeled BUSY. To check the busy flag, we merely test it to see if it is zero (not busy) or not zero (busy). The coding to do this is shown below:

```
        LDA  BUSY       Load A-Register with BUSY
        SZA             Skip the next instruction if BUSY=Ø
        JMP  *-2        Loop if driver is busy
```

5.1.3 Set the Driver Busy Flag

Upon entering the driver, we want to set the busy flag so that the driver cannot be recalled while we are using it, so we write:

```
        ISZ  BUSY
```

5.1.4 Store Calling Parameters in a Buffer Area

If you're using the .ENTR routine, calling parameters are transferred to the driver before the busy flag is checked. This necessitates the transfer of the calling parameters to a buffer area. Remember that the *address* of the calling parameter is transferred, *not the contents* of that address. Two methods are used to store parameters in a buffer area, depending upon whether we want the address of the parameter or the contents of the address of the parameter. To store the address of a calling parameter, such as a data array, we would write:

```
LDA  IDATA     Loads A-Register with address of the data array
STA  DATA      Stores this address in core location DATA
```

To store the *contents* of a calling parameter, we would write:

```
LDA  INUMB,I   Loads A-Register with the contents of INUMB
STA  NUMB      Stores these contents in core location NUMB
```

You will probably notice we are using a programming convention whereby we name the transferred parameters with an "I" prefix and the buffered parameters with their actual name. We recommend this or a similar procedure because it helps others using your drivers to understand them more easily.

5.1.5 Establish a "Reading Taken Indicator"

With interrupt drivers you are returned to the calling program immediately, long before a reading is taken; therefore, we need a method of knowing when the reading has been completed. For this purpose we set up another calling parameter "END." END is initially set to zero in the initiator section of the driver. When each reading is taken, END is incremented by one. So, as long as END=Ø, no readings have been completed. "END" is established by buffering the transferred parameter "IEND" as shown below:

```
LDA  IEND
STA  END
```

To zero this location in the initiator, we clear the A-Register and store this register in END indirectly:

```
CLA
STA  END,I
```

5.1.6 Establish Interrupt Linkage

Next we must establish the interrupt linkage, by putting an indirect JSB to the address of the continuator section of the driver in the appropriate trap cell. As you recall, the trap cell is a location in memory corresponding to a particular I/O slot. When the flag and control flip-flops are set, the computer is interrupted and forced to execute the instruction in its associated trap cell, which in the case of interrupt drivers is a JSB to the continuator section of the driver. To establish the linkage, we need the following definitions:

First, we define and label (IJSB) a jump subroutine indirect instruction with the following instruction:

```
IJSB    JSB   LINK,I
```

where LINK contains the address of the driver continuator section and is defined, on the base page, with the following instruction:

```
        ORB           Go to base page
LINK    DEF   CONT    Stores the relocated address of CONT in a reloca-
                          table base page location labeled LINK
        ORR           Return to program location
```

where CONT is the label of the continuator section of the driver.

To place IJSB in trap cell IO, we need the following coding:

```
        LDA   IJSB    Load A-register with IJSB
        STA   IO      Store IJSB in "trap cell'
```

5.1.7 Initiate the Process

The next task of the initiator section is to tell the device, in this case an input device, to begin inputting data. The instruction necessary is exactly the same as in the non-interrupt driver. We merely set the control flip-flop and clear the flag flip-flop on the appropriate I/O slot:

```
        STC   IO,C
```

where IO was previously defined by the following statement:

```
IO      EQU   12B
```

5.1.8 Return Control

The final task of the initiator is to return control to the calling program:

```
        JMP   GPR,I
```

where GPR, the entry point of the driver's initiator, contains the address of the return point of the calling program.

5.1.9 Review of Initiator

Briefly, let's review the functions that are performed by the initiator section of an interrupt driver:

1. Transfer parameters
2. Check the busy flag
3. If necessary wait till not busy, then set the BUSY flag
4. Buffer the parameters
5. Establish and zero a "reading taken" indicator
6. Establish the interrupt linkage
7. Initiate the process
8. Return to the calling program

The initiator section of a driver for the GPR is as follows:

```
ASMB,B,L,R               Control statement
        NAM  GPR         Name of program
        EXT  .ENTR       External location
        ENT  GPR         Entry point
IDATA   NOP       )
IEND    NOP       |
GPR     NOP       }      Parameter transfer
        JSB  .ENTR|
        DEF  IDATA)
        LDA  BUSY )
        SZA       }      Checks busy flag and waits while driver is busy
        JMP  *-2  )
        ISZ  BUSY        Sets busy flag
        LDA  IEND )
        STA  END  }      Buffers parameters
        LDA  IDATA|
        STA  DATA )
        CLA       }      Zeros the reading indicator
        STA  END,I)
        LDA  IJSB }      Establishes interrupt linkage
        STA  IO   )
        STC  IO,C        Initiation of process
        JMP  GPR,I       Return to calling program
IJSB    JSB  LINK,I)
        ORB        }     Establishes interrupt linkage
LINK    DEF  CONT  |
        ORR        )
(Continuator Section)
CONT    NOP
         :
*End of Driver
```

5.2 CONTINUATOR SECTION

The continuator section of the driver services the interrupt. This section will not be called directly, it will be reached only when an interrupt occurs. It is much more application oriented than the initiator section. For example, some drivers might only bring in 16 bits of data, whereas others might also convert the data from BCD to binary and check to see if the data is valid, etc. There are, however, some general functions common to all which we will discuss. These general functions are:

1. Save the registers to be used
2. Input or output data
3. Clear the busy flag
4. Bump the counters
5. Restore the saved registers
6. Return control to the point of interrupt

Before we proceed with the continuator, the following *must* be understood: the continuator *must be written in Assembler language and cannot contain FORTRAN or ALGOL subroutines, even those specially written for the continuator routine.* Since the continuator is given control with other units also possibly waiting for service, its execution time should be kept as brief as possible.

5.2.1 Continuator Entry

The first statement of the continuator section is its entry point:

```
CONT    NOP
```

The entry need not and should not be defined as an entry point with the ENT instruction. In this respect, CONT is like any other label within a subroutine. The CONT entry point would have to be defined with the ENT instruction only if it was to be called by another program. The only communication with the CONT Location is the JSB instruction located in the appropriate trap cell. Also, if we do not define CONT with an ENT instruction we eliminate the need for unique labels.

5.2.2 Saving the Registers

Saving the registers is a very simple process: we merely store them in a defined area of storage. A handy naming convention is:

SAVEA – A-Register
SAVEB – B-Register
SAVEO – E & O-Registers

The coding necessary to save all the registers is:

```
        STA   SAVEA     Save A-Register
        STB   SAVEB     Save B-Register
        ERA , ALS       E-Register to A15, bit A0=0
        SOC             Test overflow bit
        INA             Bit 1=1 if O/F set, 0 if O/F not set
        STA   SAVEO     Save the E & O-Registers
```

You have to save the E & O-Registers if you are performing any arithmetic calculations within the continuator routine, or if an INA instruction is used in the continuator. The storage areas are defined with NOP's:

```
SAVEA  NOP
SAVEB  NOP
SAVEO  NOP
```

You have to save only those registers you will be using, so in the case of the GPR example, we need only save the A-register:

```
        STA   SAVEA
```

We also have to define SAVEA somewhere in our program as follows:

```
SAVEA  NOP
```

5.2.3 Input or Output Data

Inputting or outputting data is performed with any of the following instructions:

```
        LIA   IO        Load into A-Register
        LIB   IO        Load into B-Register
        MIA   IO        Merge into A-Register
        MIB   IO        Merge into B-Register
        OTA   IO        Output the A-Register
        OTA   IO        Output the B-Register
```

In the case of the GPR driver we will simply load into the A-Register using the following instruction:

```
        LIA   IO
```

It is at this point in your continuator that you would use conversion routines, validity checks and so on. In our example, we will merely store the data as is, back into the calling program:

```
        STA   DATA,I
```

5.2.4 Increment the Reading Taken Indicator

The next order of business is to tell the main program that a data transfer has taken place. We do this by merely incrementing the core location with address "END".

```
ISZ    END,I
```

5.2.5 Zero the Busy Flag

If this is the last data transfer to take place (interrupt drivers usually ask for a number of data transfers), we must zero the location called "BUSY." We do this by clearing the A-Register and storing it in BUSY:

```
CLA
STA BUSY
```

5.2.6 Restoring the Registers

Restoring the registers is straightforward. The only word of caution is that you should restore the E & O Registers before you restore the A-Register because you use the A to restore the E & O. The coding is as follows:

```
LDA SAVEO      Loads the save word
CLO            Clears the overflow bit
SLA, ELA       Checks bit Ø and moves A15 to E-bit
STO            Sets the overflow bit if A0=1
LDA SAVEA      Restores A-Register
LDB SAVEB      Restores B-Register
```

5.2.7 Clear the Control Flip-Flop

Another task we must perform the final time we use the continuator section is to clear the control flip-flop. This enables a lower priority device to interrupt. To clear this control flip-flop, we write:

```
CLC IO
```

5.2.8 Returning Control to Interrupt Point

It is simple matter to return control to the pre-interrupted state. We merely jump indirect through the starting point of the continuator section of the driver:

```
JMP    CONT,I
```

5.2.9 Review of Continuator

That concludes the discussion of the continuator section of an interrupt driver. It is time now to study the continuator for our GPR example:

```
CONT    NOP
        STA   SAVEA       Saves A-Register
        LIA   IO          Inputs data
        STA   DATA,I      Stores data
        ISZ   END,I       Bumps indicator
        CLA           }
        STA   BUSY    }   Zeros busy flag
        LDA   SAVEA       Restores A-Register
        CLC   IO          Clears control flip-flop
        JMP   CONT,I      Returns control
SAVEA   NOP           }
BUSY    NOP           }   Defines locations
```

5.2.10 ABORT Routine

It is a good idea, especially if you are using a system that has a clock, to add an ABORT or CLEAR routine to your driver. This routine will enable you to abort a driver request if it has taken too long to complete. This is especially useful if you are using the DACE system (Data Acquisition and Control Executive) which may be performing many and varied tasks. If one piece of hardware is down, you continue to use the remaining "good" hardware. The ABORT routine is simply:

```
        ENT   ABORT
ABORT   NOP
        JSB   .ENTR       Establishes return address
        DEF   ABORT
        CLC   IO          Clears control flip-flop to avoid erroneous interrupts
        CLA           }
        STA   BUSY    }   Clears the driver's busy flag
        JMP   ABORT,I     Returns
```

5.3 REVIEW OF INTERRUPT DRIVERS

Let's review the main points of interrupt drivers:

1. They contain two sections: the initiator and the continuator.

2. The initiator:
   a. Transfers parameters
   b. Checks the driver busy flag and waits if busy
   c. Sets the busy flag

d. Buffers the parameters
e. Establishes the interrupt linkage
f. Establishes a reading indicator
g. Initiates the process
h. Returns control to the calling program

The initiator may also perform the following additional functions:

i. Configures the driver (card can be in any I/O slot and the address transferred as an argument of the call)
j. Set up counters for multiple data transfers
k. Do anything any Assembler language program can do.

3. The continuator performs the following functions:
   a. Stores the registers to be used
   b. Inputs the data
   c. Increments "reading taken" indicator
   d. Stores the data in the calling program
   e. Clears the busy flag when all data has been transferred
   f. Restores the registers
   g. Clears the control flip-flop
   h. Returns control to the pre-interrupted point

   The continuator may also perform the following additional functions:

   i. Increment and check multiple data transfer counters
   j. Re-initiate data transfer

4. The initiator may use any or all of the FORTRAN/ALGOL library routines, whereas, the continuator may *not* use any of these routines.

The complete GPR driver is shown below.

```
ASMB,B,L,R,T                  Control statement
        NAM  GPR              Name of subroutine
        EXT  .ENTR            External location
        ENT  GPR,ABORT        Entry points
IO      EQU  12B              Defines I/O slot
IDATA   NOP
IEND    NOP
GPR     NOP         }         Called point of driver
        JSB  .ENTR  }         Called point of driver parameter transfer
        DEF  IDATA  }
        LDA  BUSY   }
        SZA         }         Checks the busy flag and waits till not busy
        JMP  *-2    }
        ISZ  BUSY             Sets the busy flag
```

```
         LDA  IDATA    )
         STA  DATA     }  Buffers parameters
         LDA  IEND     }
         STA  END      )
         CLA           }
         STA  END,I    }  Zeros reading indicator
         LDA  IJSB     }
         STA  IO       }  Places IJSB in trap cell
         STC  IO,C        Initiates the device
         JSB  GPR,I
IJSB     JSB  LINK,I      Defines interrupt linkage
         ORB              Go to base page
LINK     DEF  CONT        Full address of CONT
         ORR              Return to program point
CONT     NOP              Continuator interrupt point
         STA  SAVEA       Store A-Register
         LIA  IO          Input from I/O card
         STA  DATA,I      Store data in calling program
         ISZ  END,I       Bumps reading indicator counter
         CLA           }
         STA  BUSY     }  Clear busy flag
         LDA  SAVEA       Restore A-Register
         CLC  IO          Clears control flip-flop
         JMP  CONT,I      Return control to calling program
BUSY     NOP           )
DATA     NOP           }  Define storage locations
END      NOP           }
SAVEA    NOP           )
ABORT    NOP              Abort routine
         JSB  .ENTR    }
         DEF  ABORT    }  Establishes return address
         CLC  IO       )
         CLA           }  Clears the control and busy flag
         STA  BUSY     )
         JMP  ABORT,I     Returns
         END
```

The above driver, although written for a general purpose register, will work with the photoreader. You will gain valuable additional experience by punching a tape of the driver listed above, along with a FORTRAN routine to call it and actually put the driver to work. A suitable FORTRAN routine, flowcharted in Figure 5-1, follows:

```
FTN,B,L,PROGRAM EXAM 2
 20 IF (ISSW(1)) 10,20
 10 CALL GPR (IDATA,IEND)
 40 IF (-IEND) 30,40
 30 WRITE (2,100) IDATA
100 FORMAT ("DATA IS" I6)
    GO TO 20
    END
    END$
```

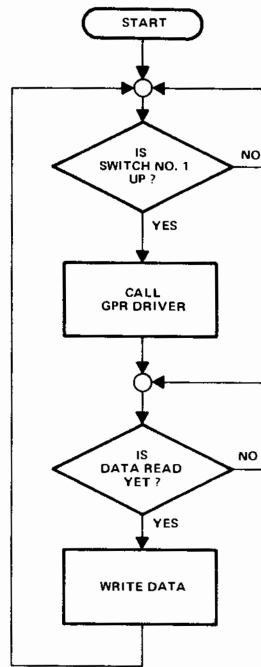


Figure 5-1. Data Read-Write Program Flowchart

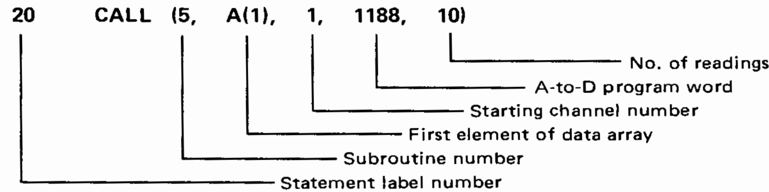
Once you have that driver working, modify it so that it will take multiple readings. This requires you to transfer an additional parameter called "NUMB," the number of readings to be taken. There are a few games you have to play, so go ahead and try. Happy driver writing!

VI. ABSOLUTE DRIVERS FOR BASIC

HP BASIC provides a special statement, the CALL statement, for controlling instruments or special devices in BASIC language. The CALL statement provides a means of transferring control to, and exchanging parameters with, absolute Assembler language input/output driver subroutines which are added to the standard BASIC compiler system. The CALL statement has the following general form:

CALL (< subroutine number > , < parameter list >)

A representative call might be as follows:



Drivers for programming instrumentation from the BASIC language must be written in absolute Assembler language because they have to be integrated precisely with the BASIC compiler, which they modify. The absolute drivers differ from the previously-described relocatable non-BCS drivers in that:

1. They usually are non-interrupt drivers.
2. They cannot utilize the relocating loader.

This section will demonstrate the necessary requirements of BASIC drivers. We will be using the GPR card again as an example. This will enable you to compare relocatable non-BCS drivers with absolute drivers. This section will begin with a discussion of the problems encountered when writing absolute programs, proceed with the structure of the BASIC compiler, and end with the GPR card example. *Because you will be working intimately with the compiler, HP recommends that you order listings of your BASIC compiler, and the Prepare Basic System which is used to configure it, so that these will be available for reference.*

6.1 PAGING

When writing absolute routines, the programmer must be concerned with paging problems. As you recall, the memory of HP computers is divided into "pages." Each page contains 1000 words of core, so in an 8K machine we will have a "base page" plus 7 additional pages as in Figure 6-1, overleaf.

	OCTAL ADDRESSES
PAGE 7	17777 16000
PAGE 6	15777 14000
PAGE 5	13777 12000
PAGE 4	11777 10000
PAGE 3	7777 6000
PAGE 2	5777 4000
PAGE 1	3777 2000
BASE PAGE	1777 0

Figure 6-1. Division of 8K Memory into Pages

A memory reference instruction such as a load into A-Register (LDA) instruction can reference any location on its own page or the base page only. For example, an instruction on Page 3 can load into A only memory locations 0-1777 and 6000-7777. To load into A from any other core location, we must use indirect addressing. For example, to load into A from address XX, we write LDA XX, I, where

XX contains the address of the core location we wish to load into A.

The reason we did not have to worry about paging before is because the relocating loader took care of establishing the above indirect links for us. When we write absolute programs, we must worry about these ourselves.

Paging will not be a problem when all our drivers will fit on one page. To determine if this is true, we must actually compile the driver using the assembler and then check the listing to see if a page boundary has been crossed. If so, you must establish the necessary indirect links or rearrange your drivers to fit on one page. An indirect link example will be shown later.

CAUTION: The locations referenced throughout this section are true only for the HP 20112A version of the BASIC compiler. The addresses will be different for other versions of the BASIC compiler. Addresses needed for writing absolute drivers for other versions of the BASIC compiler are listed in Table 6-1 at the end of this section

6.2 LOCATING YOUR DRIVER

Since we are writing absolute, we must originate the driver at some core location by using the ORG instruction. To determine this location, you should have a listing of your BASIC compiler. For most applications, you will want to overlay the matrix routines. To find the correct start point in your listing of the BASIC compiler, look for the headline and statements shown in the following excerpt from the listing of the HP 20112A BASIC compiler.

```
 PAGE 0119 #08  MATRIX ROUTINES


0317*                  ******************************
0318*                  MATRIX STMT EXECUTION CONTROL
0319*                  ******************************
0320  11436 160307  EMAT  LDA TEMPS,I   LOAD FIRST WORD OF STMT
0321  11437 010260        AND MSK1      ISOLATE OPERAND PART
0322  11440 002003        SZA,RSS       IF NULL , MUST BE PRINT OR
0323  11441 027547        JMP M001      READ , JUMP TO FURTHER CHECKS
0324  11442 014507        JSB SSYMT     SEARCH FOR OPERAND IN SYMTAB
0325  11443 006004        INB
2326  11444 077543        STB MCAL+2    STORE SYMTAB ADDRESS IN CALLING
```

where the 11436 in the second column opposite EMAT is the octal address we are looking for.

If you do overlay the matrix routines, you must also modify the BASIC compiler so that an error is printed out if the matrix routine is referenced. This is accomplished by placing LET instructions in those locations of the "Name Table for Operators" that define the function MAT. In the HP 20112A BASIC compiler, these locations are 4063_8, 4064_8, 4065_8. To find these locations in other versions of the BASIC compiler, look in the "CHECK SYNTAX AND TRANSLITERATE" section for the "Print Name Table for Operators." Then locate the address of MAT, shown in the following sample printout.

```
 PAGE 0049 #04  CHECK SYNTAX AND TRANSLITERATE


0701  04052 054005  INPUT OCT 54005
0702  04053 044516        ASC 3,INPUT
0703  04056 055007  PSTOR OCT 55007
0704  04057 051105        ASC 4,RESTORE
0705  04063 056003  MAT   OCT 56003
0706  04064 046501        ASC 2,MAT
0707  04066 057004  THEN  OCT 57004
```

To disable the matrix routines of the HP 20112A compiler and set the starting address to overlay them with our drivers, we do the following:

```
        ORG  4063B
LET     OCT  32003
        ASC  2,LET
        ORG  11436B
```

and thus we have the first four statements of a driver or drivers that overlay the matrix routines of the HP 20112A version of the BASIC compiler.

If you wish to retain the matrix routines because you use them elsewhere in your program, you must originate your drivers at some point behind the matrix routines (usually immediately behind). To find the first available position of storage is a simple matter because that information is contained in octal core location 110_8. 110_8 contains the address of the first word of program available storage.

There are two ways to get the contents of 110_8. The first is to load your BASIC compiler, load address 110_8, display memory, and read the data in the M-Register. The second method is to check the location of FINIS in your listing of the BASIC compiler (position in HP 20112A listing is shown below):

```
 PAGE 0144 #09  MATRIX ROUTINES

0835  13371 000000  LPIV  BSS 1
0836  13372 114374  LPLUS JSB .FADA,I   GENERATES CODE
0837  13373 013360        DEF T18
0838  13374 036050  INCB1 ISZ B1        GENERATES CODE
```

```
0844  13402 013362  LAD1  DEF T20
0845  13403         SBTBL EQU *
0846  13403         LSBTB EQU *
0847  13403         FINIS EQU *
0848  00110               ORG FWAM
0849  00110 013403        DEF FINIS
0850  07457         TT1   EQU .FDV
0851  07546         TT2   EQU IDIV
0852  00313         TT3   EQU TEMPS+4
0853  00314         TT4   EQU TEMPS+5
0854  10355         FFLAG EQU ETAN
0855                      END
**  NO ERRORS*
```

The address of FINIS will be the same as the contents of 110_8. It is the first word of available memory.

Since we are not overlaying any part of the BASIC compiler, the only coding we need to originate at the first available word of memory in the HP 20112A version is:

```
ORG 13404B
```

6.3 LINKAGE TABLE

After we originate our drivers at the appropriate core location, we must set up the linkage table. The linkage table supplies the subroutine calling number, the number of parameters to be passed, and the address of the entry point of the driver. This information is used by the BASIC compiler to determine if a called subroutine exists, if the correct number of parameters have been specified, and where the driver is located. It is important to note that each driver we wish to add to the BASIC language must have its own entry in the linkage table.

Entries in the subroutine linkage table, one per subroutine, are two words in length. The first word contains the calling number of the subroutine (any number from 1 to 77_8 inclusive) and the number of parameters to be passed, in the format of Figure 6-2.

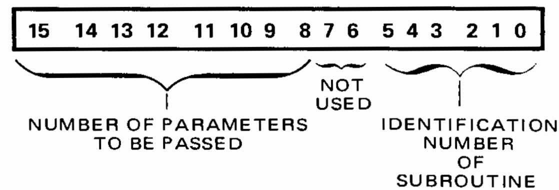


Figure 6-2. Format of First Word of Linkage Table Entry

The second word of the table entry contains the address of the entry point of the driver. This is the label of the driver. It is defined with a DEF instruction shown below:

```
DEF LABEL
```

The first entry of the linkage table must have a label so we can refer to this address later. The label we use is SBTBL — SUBTABLE.

The coding for the subtable that links three drivers with labels GPR, DRIVE, and DO, using calling numbers 6, 17, and 1 respectively, is shown at top of page 6-6. GPR is to pass 2 parameters, DRIVE — 3 parameters, and GO — 6 parameters.

```
SBTBL   OCT  1ØØ6     Sub 6 has 2 parameters
        DEF  GPR      Names GPR driver
        OCT  1417     Sub 15 has 3 parameters
        DEF  DRIVE    Names DRIVE driver
        OCT  3ØØ1     Sub 1 has 6 parameters
        DEF  GO       Names GO driver
```

To tell the compiler that the linkage table is complete, we have to label the end of the table with:

```
ENDTB   EQU  *
```

The total coding thus far which overlays the matrix routines and establishes the linages with the compiler is:

```
        ORG  4063B  )
LET     OCT  32003  > Disables MAT Linkages
        ASC  2,LET  )
        ORG  11436B   Originates at MAT routines
SBTBL   OCT  1ØØ6   }
        DEF  GPR    } Links "GPR" Driver no. 6, passing 2 parameters
        OCT  1417   }
        DEF  DRIVE  } Links "Drive" Driver no. 15, passing 3 parameters
        OCT  3ØØ1   }
        DEF  GO     } Links "GO" Driver no. 1, passing 6 parameters
ENDTB   EQU  *        Defines End of Linkage table
```

6.4 BODY OF THE DRIVER

The next order of business is adding the driver subroutines. Later we will show you how to write the driver using a GPR card as an example. For now, let us assume that we already have the driver coded. We then place it immediately behind the linkage table as shown:

```
ENDTB   EQU  *
GPR     NOP             )
         :              > Subroutine 6
        JMP  GPR,I      )
DRIVE   NOP             )
         :              > Subroutine 15
        JMP  DRIVE'I    )
GO      NOP             )
         :              > Subroutine 1
        JMP  GO,I       )
```

These can be in any order since we pass the label to the subroutine linkage table.

6.5 ENDING SERVICE TABLE

After we have attached all our subroutines and drivers we must add the ending service table. The purpose of this table is to tell the BASIC compiler what the address of first word of program available memory is now, and the bounds of the subroutine linkage table. In the HP 20112A version of the BASIC compiler the address of the first word of available memory (FWAM) is located at 110_8 and the bounds of the linkage table are at 121_8 (ASBTB) and 122_8 (SBTBE) as shown below:

```
 PAGE 0002 #01

0001                   ASMB,A,B,L
0002  00100                  ORG 100B
0003                         SUP
0004  00100 124335           JMP STRTA,I     JUMP TO STARTING ADDRESS
0005  00101 000000   PREAD BSS 1             PHOTO READER LINK
2006  00102 000000   WRITE BSS 1             TTY OUTPUT LINK
0007  00103 000000   PUNCH BSS 1             PUNCH LINK
0008  00104 000000   REED  BSS 1             KEYBOARD LINK
0009  00105 000000         BSS 1             STOP                         JSB,I
0010*
0011  00106 000000         BSS 1            LINK TO INTERRUPT OFF ROUTINE
0012  00107 000000         BSS 1            LINK TO INTERRUPT ON  ROUTINE
0013*
0014  00110 000000   FWAM  BSS 1            FIRST WORD OF AVAILABLE MEMORY
0015  00111 000000   LWAM  BSS 1             LAST WORD OF AVAILABLE MEMORY
0016  00112 002000   PROGF DEF 2000B         FIRST WORD OF USERS PROGRAM
0017  00113 001777   PROGL DEF 1777B        LAST WORD+1 OF USERS PROGRAM
0018  00114 000000   TLINK BSS 1             TTY INTERRUPT LINK
0019  00115 000000   FCORE BSS 1            START OF FREE CORE
0020  00116 000000   SYMTF BSS 1            START OF SYMBOL TABLE
0021  00117 000000   SYMTA BSS 1             SYMBOL TABLE END
0022  00120 000000   LSTAK BSS 1
0023  00121 013403   ASBTB DEF SBTBL
0024  00122 013403   SBTBE DEF LSBTB         LAST WORD + 1 OF CALL TABLE
0025  00123 000000   .BUFA BSS 1            IO BUFFER ADDRESS
```

To construct the ending service table, we have to define the first word of available memory which is the memory location addressed after all drivers have been loaded. To do this, we use this EQU instruction:

```
LSTWD   EQU   *
```

Next we must place this address (LSTWD) into core location 110_8. We do this with the following:

```
        ORG   110B
        DEF   LSTWD
```

Then we must establish the bounds of the linkage table in core locations 121_8 and 122_8. From the linkage table coding discussed previously, we can pick off the starting address of SBTBL and the ending address of ENDTB. To complete the ending service table we must write:

```
        ORG  121B
        DEF  SBTBL
        DEF  ENDTB
```

The ending service table as a whole is:

```
LSTWD   EQU  *          Defines last word
        ORG  11ØB   \   Places address of first word of available memory in
        DEF  LSTWD  /     110₈.
        ORG  121B   \   Places the bounds of the subroutine linkage table
        DEF  SBTBL   >    into 121₈ and 122₈.
        DEF  ENDTB  /
```

6.6 REVIEW OF ABSOLUTE DRIVERS FOR BASIC

The organization of adding subroutine-type drivers to the BASIC language is shown below for HP 20112A version of the BASIC compiler:

```
        ORG  4063B  \
LET     OCT  32003   >  Disables matrix functions
        ASC  2,LET  /
        ORG  11436B     Overlays matrix functions
SBTBL   OCT  1ØØ6   \
        DEF  GPR     |
        OCT  1417    |
        DEF  DRIVE   |  Linkage table which passes calling number, number
         :           >    of parameters, and names of routines to BASIC
         :           |    compiler.
        OCT  3ØØ1    |
        DEF  GO      |
ENDTB   EQU  *      /
GPR     NOP         \
         :           |
        JMP  GPR,I   |
DRIVE   NOP          |
         :           >  User-written BASIC drivers. Up to 77₈ drivers can
        JMP  DRIVE,I |    be added.
GO      NOP          |
         :           |
        JMP  GO,I   /
LSTWD   EQU  *      \
        ORG  11ØB    |
        DEF  LSTWD   |  Ending service table which places the address of the
        ORG  121B    >    available memory in 110₈ and the bounds of the
        DEF  SBTBL   |    linkage table in 121₈ and 122₈.
        DEF  ENDTB   |
        END         /
```

CAUTION: Please keep in mind that all the addresses referred to above are from the HP 20112A version of the BASIC compiler. Any of these addresses may be different in other versions of the BASIC compiler.

6.7 BASIC DRIVERS

BASIC drivers are usually non-interrupt drivers. This simplifies the task of preparation because we no longer have to provide for recurrent calls, storing and restoring registers, trap cells, and so on. The only functions we have to be concerned with are:

1. The location of the driver.
2. Transferring parameters
3. Converting some parameters to integer form
4. Inputting or outputting data
5. Returning to the calling program.

6.7.1 Location of the Driver

Most drivers will be written individually instead of being written in groups as required by the organization of BASIC. To develop a BASIC driver, you merely ORG at the core location where you wish to place it. The location depends upon what and where other drivers are located in the system as well as the length of the linkage table, etc. The first statement of your driver will be an ORG statement:

```
ORG  14500B      Originate at core location 14500₈
```

If you assemble all your drivers as a group, you can eliminate this statement at the start of each driver; it is required only at the start of the first driver in the group.

6.7.2 Transferring Parameters

There is no library routine, such as .ENTR, that we can use to transfer parameters. To help you out, however, we have rewritten .ENTR to be compatible with BASIC. The listing and discussion of this routine is contained in Appendix B. As in the case of .ENTR, you need not understand the routine in order to use it. From your viewpoint, it looks exactly like the .ENTR routine of the FORTRAN/ALGOL Library. You merely define the parameters to be transferred by NOP's:

```
IDATA  NOP
IIO    NOP
```

Then place the entry point of the driver (the same name as used in the subroutine linkage table).

```
GPR     NOP
```

Followed by a jump subroutine to .ENTR:

```
        JSB   .ENTR
```

If .ENTR is not on the same page as the driver requesting it, a base page link must be established. To do this, go to the base page:

```
        ORG  77B
```

Then establish the link:

```
ENTR    DEF  .ENTR
```

In our driver we now have to jump to .ENTR indirectly by:

```
        JSB   ENTR,I
```

Followed by the address of the first parameter to be transferred, which in this case was IDATA:

```
        DEF  IDATA
```

As a whole, the parameter transfer routine is:

```
IDATA   NOP               Defines data
IIO     NOP               Defines I/O slot
GPR     NOP               Names driver
        JSB   ENTR,I      Jumps to parameter routine indirectly
        DEF   IDATA       Passes address of first parameter
```

It is important to note two poinsts:

1. the *addresses* of the parameters are transferred and not the contents of the parameters.
2. in BASIC all numbers are represented in floating point form.

The first situation, that of transferring the address, is handled almost as it was in the case of non-BCS relocatable drivers. To obtain the address of the first word of a parameter, we merely load the A-register. The second situation complicates the process because each parameter in floating point form occupies two words of memory. This requires that we load the A-register indirectly with the first word of the parameter and the B-register indirectly with the second word; then, depending on the parameter, we may want to change this to an integer number as would be the case with the channel number of a scanner.

6.7.3 Base Page Links

You can put your base page links anywhere between memory locations 10_8 and 77_8, provided that the location is not being used as a 'trap cell'. In other words, use those memory locations $10_8 - 77_8$ that don't have I/O cards plugged into corresponding slots. For instance, if you have Teletype, photo reader, and tape punch interface cards in slots 10_8, 11_8, and 12_8 and all other interface card slots are empty, you can use memory locations $13_8 - 77_8$ for base page links.

6.7.4 Conversion Between Floating Point and Integer Forms

To convert from floating point to integer form, we use the IFIX routine. IFIX is part of the BASIC compiler. To access this routine, we first must obtain the address of it in the BASIC compiler. In the HP 20112A version of BASIC, it is located at 1477B as shown below:

```
PAGE 0019 #02

0139**
0140***  INTEGERIZE FLOATING POINT NUMBER  **
0141**
0142  01477 000000  IFIX  NOP            NUMBER IN (A) AND (B)
0143  01500 071435        STA GETCR      SAVE MANTISSA
0144  01501 015531        JSB .FLUN      UNPACK LOW WORD
0145  01502 040237        ADA M16        COMPUTE SHIFT COUNT
0146  01503 002021        SSA,RSS        EXP >= 16?
0147  01504 025523        JMP IFIX3      YES
0148  01505 103101        CLO            NO, SET
0149  01506 006002        SZB              OVERFLOW IF
```

We obtain the address by using the EQU instruction:

IFIX EQU 1477B

To call the routine, since it is on the base page, we can jump subroutine directly to it:

```
JSB   IFIX
```

This routine converts what is contained in the A & B registers to integer form and returns the value in the A-register.

The coding to access a parameter and convert it to integer form is:

```
LDA  ICHAN,I    Load A with 1st word of channel number
ISZ  ICHAN      Add 1 to the address of ICHAN
LDB  ICHAN,I    Load B with 2nd word of channel number
JSB  IFIX       Jump to the IFIX routine
STA  CHAN       Store result in CHAN
```

where ICHAN is the address of the core location where the channel number of the scanner is stored and CHAN is the driver's address of where the channel number is to be stored.

When you take data and want to store it back into the calling program, you must, if it is an integer, convert it to floating point form. To perform this function, we use the FLOAT routine. Since the FLOAT routine is not on the base page, we must set up an indirect link to access it. To do this, we define FLOAT as an octal equal to the compiler's address of the FLOAT routine:

```
FLOAT  OCT  11370
```

The address 11370B was obtained from the HP 20112A version of the BASIC compiler as shown below:

```
PAGE 0115 #08  LIBRARY ROUTINES

0231*                     ******************************
0232*                     SUBROUTINE TO FLOAT AN INTEGER
0233*                     ******************************
0234*
0235*                     CALLED BY JSB FLOAT WITH INTEGER IN A
0236*                     THE FLOATING POINT EQUIVALENT IS RETURNED
0237*                     IN A & B
0238*
0239   11370 000000   FLOAT NOP
0240   11371 064146         LDB .15
0241   11372 074276         STB EXP
0242   11373 006400         CLB
0243   11374 015267         JSB .PACK
0244   11375 127370         JMP FLOAT,I
```

To convert an integer to floating point, we must jump subroutine indirect to FLOAT:

```
JSB   FLOAT,I
```

The process of returning a number to the calling program is shown below:

```
LDA   NUMB       Load A with the number
JSB   FLOAT,I    Jump to float routine
STA   DATA,I     Store word 1 in data
ISZ   DATA       Increment address of data
STB   DATA,I     Store word 2 in data
```

6.7.5 Inputting and Outputting

Inputting and outputting in BASIC drivers is an non-interrupt process. The reason for this is a characteristic of the BASIC Language. At various points within the BASIC compiler, the interrupt system is turned off. If you receive an interrupt when the interrupt system is disabled, you lose it. What this means is that if you write interrupt drivers for BASIC, you can lose data from synchronous devices.

To write non-interrupt drivers, we return to the method discussed first in this booklet, using the skip if flag set (SFS) instruction, as shown below:

```
STC   IO,C       Initiates device
SFS   IO         Tests flag
JMP   *-1        Loop if not ready
LIA   IO
```

where IO is the number of the I/O slot serving the selected device.

In order to use the SFS command, the interrupt system must be bypassed. To do this, we place a clear control command in the "trap cell" assigned to the selected device. Now, when an interrupt occurs, the CLC is executed, and control is returned to the main program. The flag flip-flop remains set so we can test it with the SFS instruction. The coding is simply:

```
LDA   CLC
STA   IO
```

where CLC is defined by:

```
CLC      CLC   IO
```

6.7.6 GPR Driver

The complete absolute driver for the General Purpose Register card, to bring in 16-bits of data for use by the calling program in floating point binary form is as follows:

```
        ORG  15000B       Originates program at predetermined point
IO      EQU  12B        } Defines parameters
IDATA   NOP             }
GPR     NOP               Labels driver
        JSB  .ENTR      } Transfers parameters (Assumes .ENTR is on the
        DEF  IDATA      } same page)
        LDA  CLC        } Places a CLC in device's trap cell
        STA  IO         }
        STC  IO,C       }
        SFS  IO         } Initiates the device and waits for the reading
        JMP  *-1        }
        LIA  IO           Inputs the data
        JSB  FLOAT,I    }
        STA  IDATA,I    } Converts the data to floating point and stores it in
        ISZ  IDATA      } the calling program
        STB  IDATA,I    }
        JMP  GPR,I        Returns control
FLOAT   OCT  11370        Defines FLOAT
CLC     CLC  IO
        END
```

6.7.7 Review of BASIC Drivers

Following are the main parts of absolute BASIC drivers:

1. Originate them at the proper point.
2. Establish the linkage table.
3. Locate the drivers in memory.
   a. Transfer parameters
   b. Convert these parameters
   c. Place a CLC in device's trap cell
   d. Initiate device and wait for reading
   e. Convert reading to floating point and store in calling program.
   f. Return to calling program
4. Establish the ending service table.
5. Check for problem areas.
   a. Paging requirements
   b. Array sizes
   c. Locations used from the BASIC compiler
   d. Appropriate calling sequence

Shown below is the GPR driver we just developed integrated into the HP 20112A version of BASIC with the .ENTR routine attached. In this example, we are overlaying the MATRIX routines:

```
ASMB,A,L,B,T                  Requests an absolute program
       ORG 4063B      )
LET    OCT 32003      )
       ASC 2,LET      )  Overlays and disables matrix routines
       ORG 11436B     )
SBTBL  OCT 401        )
       DEF GPR        )  Linkage table
ENDTB  EQU *          )
IO     EQU 17B        )
IDATA  NOP            )
GPR    NOP            )
       JSB ENTER,I    )
       DEF IDATA      )
       LDA CLC        )
       STA IO         )
       STC IO,C       )
       SFS IO         )
       JMP *-1        )  Body of GPR driver
       LIA IO         )
       JSB FLOAT,I    )
       STA IDATA,I    )
       ISZ IDATA      )
       STB IDATA,I    )
       JMP GPR,I      )
IFIX   EQU 1477B      )
FLOAT  OCT 11370      )
CLC    CLC IO         )
.ENTR  NOP            )
       STA SORCE      )
       LDB .ENTR,I    )
       STB DEST       )
       CMB,INB        )
       ADB .ENTR      )
       ADB M2         )
       CMB,INB        )
       STB CNTR       )
       ISZ .ENTR      )
LOOP   LDA SORCE,I    )
       STA DEST,I     )  Parameter transfer routine
       LDA SORCE      )
       ADA M1         )
       STA SORCE      )
       ISZ DEST       )
       ISZ CNTR       )
       JMP LOOP       )
       JMP .ENTR,I    )
M1     DEC -1         )
M2     DEC -2         )
SORCE  NOP            )
DEST   NOP            )
CNTR   NOP            )
```

(Continued overleaf)

```
LSTWD EQU  *
      ORG  110B
      DEF  LSTWD      Ending service table
      ORG  121B
      DEF  SBTBL
      DEF  ENDTB
      ORG  77B
ENTER DEF  .ENTR      Base page link
      END
```

6.7.8 Integrating the Driver into BASIC

To integrate the GPR driver into BASIC, you merely have to load a compiled tape of it after you load the BASIC compiler. The sequence of events is as follows:

1. Load address 177ØØ (8K) or 377ØØ (16K), enable the loader.
2. Place configured BASIC compiler in the photoreader and push "RUN."
3. When "Halt 1Ø2Ø77" occurs, place the BASIC driver in the photoreader and push "RUN."
4. When "Halt 1Ø2Ø77" occurs, protect the loader, load address "ØØØ1ØØ" and push "RUN." The teletype should respond by typing "READY."

6.7.9 Using the Driver

You are now ready to write a BASIC program that utilizes the above driver. A very simple example to input five data points and print them is shown below:

```
READY
1ØFORI=1TO5
2ØCALL(1,D1)
3ØPRINT D1        Program body
4ØNEXTI
5ØEND
RUN
2
211
4                 Results
24
44
READY
```

6.8 PROTECTING THE BASIC COMPILER

The BASIC language is not protected when you add drivers to it. Whenever data is transferred from a called subroutine through the address of a parameter, there is a possibility that the user's program that calls the subroutine, or even the BASIC system itself, might be destroyed. To assure that intended results are obtained, you should be aware of the following precautions that can be taken for avoiding possible sources of trouble.

6.8.1 Calling Parameters That Are Expressions

A calling parameter that is an expression (e.g., A+B, A and B) will be evaluated and the result placed in a temporary location. Since the parameter address references this temporary address, storing into it will not harm the BASIC system, but the value stored there is lost to the BASIC program. Using labels set equal to expressions in place of the expression itself will eliminate this problem.

6.8.2 Overload Array Allocation

If a subroutine stores more values in an array than the array can hold, the resulting overflow of data can destroy the BASIC system. DIMENSION statements or COMMON statements in the user's program must set aside sufficient storage for the number of data values to be taken by the subroutine during one call. Software checks whereby the dimensions of the array are passed as additional calling parameters, such as the number of readings to be taken, help to prevent overflow.

6.8.3 Unsuitable Calling Parameters

Calling parameters that are unsuitable, such as a constant where the driver is expecting an array parameter, can destroy the BASIC system. The programmer should be sure that he is sending suitable parameters to the subroutine.

6.8.4 Checking Storage in the Driver

Really effective protection can be written into the driver at the cost of additional programming effort. BASIC contains sets of pointers that delimit areas of memory within which different types of parameters can be stored, as shown in Figure 6-3.

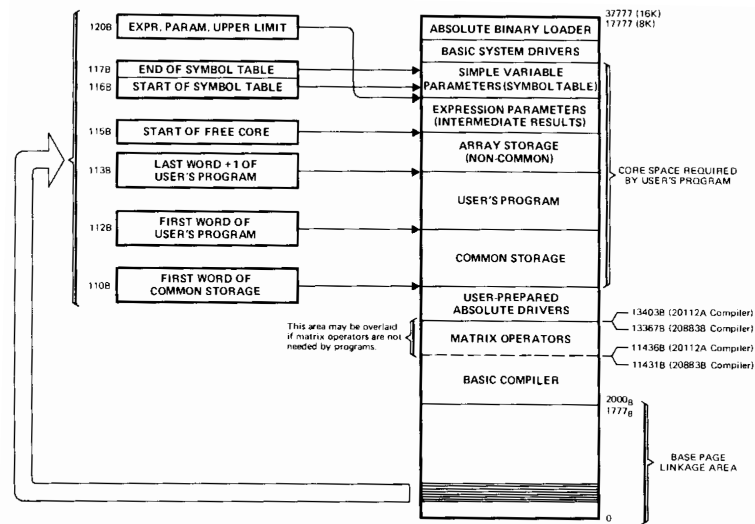


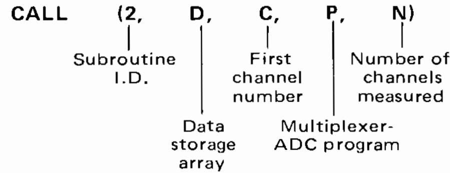
Figure 6-3. How Pointers Delimit Parameter Storage Areas in Core

By checking parameter addresses against the appropriate limits, the driver subroutine can confirm that storage has been allocated in the appropriate storage areas for the various parameters. If X represents the parameter address, the following apply:

1. For a constant: (Addr. in 112B) $<$ X $<$ (Addr. in 113B).
2. For a simple variable: (Addr. in 116B) $<$ X $<$ (Addr. in 117B).
3. For an array:
   a. In common storage: (Addr. in 110B) $<$ X $<$ (Addr. in 112B).
   b. In non-common storage: (Addr. in 113B) $<$ X $<$ (Addr. in 115B).
4. For an expression: (Addr. in 115B) $<$ X $<$ (Addr. in 120B).

where 112B, etc. designates pointer address location 112_8 in memory.

To illustrate the use of storage checking, suppose that you are writing a driver for BASIC that will program a data acquisition subsystem to take 'N' channels of data, in response to this CALL from a BASIC program:



Our driver is written to convert data to floating point, in which form two memory locations are required for storage of each data point. Let's assume that addressing of the data to storage is provided by a variable pointer, designated MBUF. The initial address in MBUF is transferred from the BASIC program in the CALL to the driver. Thereafter, data is stored in the first and second addresses of each data point, with MBUF incremented as necessary, until 'N' data points have been measured and stored, as specified in the CALL.

The principal danger with this type of driver is that the data it brings in may overflow storage, particularly where the user-programmer has not set aside sufficient array storage for the data with a COM or a DIM statement at the start of his BASIC program. Another danger is that the programmer's CALL may contain transposed parameters, with D possibly not in the first position. In either instance of error, the storage check subroutine in the driver can prevent overflow and alert the user.

The first step in implementing the storage check subroutine is to define the bounds against which the MBUF addresses are to be checked. These definitions would immediately follow the ORG statement at the start of the driver:

```
PROFG  EQU  112B    First address of user's program is in 112B.
PROGL  EQU  113B    Last address of user's program is in 113B.
FCORE  EQU  115B    First address of free core is in 115B.
```

Prior to storage of each data point, a JSB to the following SCK (storage check) subroutine imbedded in the driver can be used to determine if the data will overflow the specified bounds:

```
SCK    NOP          Entry point (with MBUF in B-register)
       INB          Increment to get second half address of data point.
       STB   TEMP   Temporarily store address.
```

```
         LDB   PROGF  )
         CMB , INB    (  Is MBUF address in common storage, below the ad-
         ADB   TEMP   (  dress of the first word of the user's program?
         SSB          )
         JMP   SCK,I     Yes, return to main sequence and store data.
         LDB   PROGL  )
         CMB , INB    (  Is MBUF address in user's program?
         ADB   TEMP   (
         SSB          )
         JMP   SERR      Yes, jump to storage error message print routine.
         LDB   FCORE  )
         CMB , INB    (  Is MBUF address in non-common storage, below the
         ADB   TEMP   (  start of free core?
         SSB          )
         JMP   SCK,I     Yes, return to main sequence and store data.
         JMP   SERR      No, jump to storage error message print routine.
          :
SERR     LDA   .46
         LDB   SCKBF
         JSB   102B,I
         JMP   MONIT,I   Return to BASIC program monitor (types READY).
          :
CKBUF    OCT   6412      Carriage return and line feed.
         ASC   19,       CALL ROUTINE DATA POINTER OUT OF
         ASC   4, LIMITS
SCKBF    DEF   CKBUF
```

The diagnostic message 'CALL ROUTINE DATA POINTER OUT OF LIMITS' would all be printed on one line, indented six spaces, followed by 'READY' (not indented) on the next line.

The value .46 would be defined as decimal 46, and the pointer MONIT to the entry point of the BASIC system monitor routine would be defined at the start of the driver by:

```
.46      EQU 166B   The decimal value 46 is in 166₈.
MONIT    EQU 337B   The entry address of the monitor routine is in 337₈.
```

(The above addresses are true for the HP 20112A version of BASIC, but would be different for other versions; in the HP 20883A version, for instance, decimal 46 is in location 217_8 and the address of the monitor routine entry point is in 344_8.)

Table 6-1

Address Reference Chart for Integrating Absolute Drivers into BASIC Compilers

OBJECTIVE	Coding With 20112A Compiler	Coding With 20883A Compiler
Originate driver that will overlay matrix routines. (Used if matrix routines are not needed.)	ORG 4063B LET OCT 32003 ASC 2,LET ORG 11436B	ORG 4105B LET OCT 32003 ASC 2,LET ORG 11431B
Originate driver immediately after matrix routines.	ORG 13404B	ORG 13367B
Construction of ending service table.	LSTWO EQU * ORG 110B ORG 121B DEF SBTBL DEF ENDTB	LSTWO EQU * ORG 110B ORG 121B DEF SBTBL DEF ENDTB

Conversion of data (channel number) from floating point to integer form.
NOTE: EQU can be used as at right *only* for entry points on the base page (addresses 0 through 1777B).
	IFIX EQU 1477B
	:
	LDA ICHAN,I
	ISZ ICHAN
	LDB ICHAN,I
	JSB IFIX
	STA CHAN
		IFIX EQU 1511B
		:
		LDA ICHAN,I
		ISZ ICHAN
		LDB ICHAN,I
		JSB IFIX
		STA CHAN

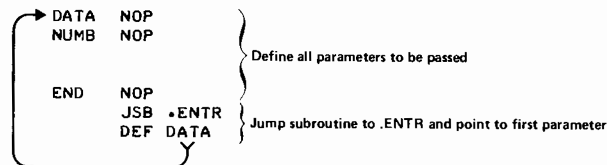
Conversion of integer to floating point form. NOTE: For entry points not on the base page, OCT (or DEF) must be used and the JSB must be indirect, as shown at right.	FLOAT OCT 11370 : LDA NUMB JSB FLOAT,I STA DATA,I ISZ DATA STB DATA,I	FLOAT OCT 11404 : LDA NUMB JSB FLOAT,I STA DATA,I ISZ DATA STB DATA,I
Pointer to First Word of Available Memory.	110B	110B
Pointer to First Word of User's Program.	112B	112B
Pointer to Last Word +1 of User's Program.	113B	113B
Pointer to Start of Free Core.	115B	115B
Pointer to Start of Symbol Table.	116B	116B
Pointer to End of Symbol Table.	117B	117B
Pointer to Last Address for Expression Parameters.	120B	120B
Start of Decimal Definitions (positive numbers)	134B	166B
Definition of decimal 46	166B	217B
Pointer to Entry Address of Monitor Routine	337B	344B


```
      ISZ  DATA       } Increment points and reading indicators
      ISZ  END,I
      ISZ  CNTR
      JMP  *+5          Repeat call if NUMB readings have not been taken
      CLA             } Clear BUSY flag
      STA  BUSY
      CLC  IO           Clear control flip-flop
      JMP  CONT,I       Return
      STC  IO,C         Re-initiate the device
      JMP  CONT,I       Return
END   NOP             } Define core locations
DATA  NOP
CNTR  NOP
BUSY  NOP
SAVEA NOP
IJSB  JSB  LINK,I     } Establish linkages
      ORB
LINK  DEF  CONT
      ORR
      END
```

APPENDIX B

.ENTR ROUTINE FOR BASIC DRIVERS

.ENTR is a routine that passes parameter *addresses* (not their contents) from the calling program to the called program. To use this routine, merely locate it in memory with your BASIC drivers when you add them to the BASIC compiler. The calling sequence from assembler language is:



The .ENTR routine is listed below:

```
.ENTR  NOP
       STA  SORCE      "A" contains address of parameter No. 1 address.
       LDB  .ENTR,I    Get address of first driver buffer location.
       STB  DEST       Store that address in a working buffer.
       CMB,INB       }
       ADB  .ENTR    } Determine the number of elements in the address array.
       ADB  M2       }
       CMB,INB       } Make that number negative and use it to initialize the transfer counter.
       STB  CNTR     }
       ISZ  .ENTR      Increment .ENTR to correct return address.
LOOP   LDA  SORCE,I    Get address of parameter.
       STA  DEST,I     Store in subroutine address array.
       LDA  SORCE    }
       ADA  M1       } Decrement pointer to next parameter address and save that address.
       STA  SORCE    }
       ISZ  DEST       Increment pointer to subroutine array.
       ISZ  CNTR       Last parameter?
       JMP  LOOP       No, process address of next parameter.
       JMP  .ENTR,I    Yes, return to execute the subroutine.
M1     DEC  -1
M2     DEC  -2
SORCE  NOP
DEST   NOP
CNTR   NOP
```

$1

5950-8943

HEWLETT *hp* PACKARD

395 PAGE MILL ROAD, PALO ALTO, CALIFORNIA 94306

Printed in U.S.A. 11/69